SQL-Tutor: a preliminary report

Antonija Mitrović
Computer Science Department
University of Canterbury
Private Bag 4800, Christchurch, New Zealand
tanja@cosc.canterbury.ac.nz

August 21, 1997

Abstract

Intelligent Tutoring Systems (ITS) are computer systems which provide students with learning environments adapted to their knowledge and learning capabilities. This paper presents SQL-Tutor, an ITS for SQL programming. SQL, the dominant database language, is a simple and highly structured language; yet, students have many difficulties learning it. SQL-Tutor is designed as a guided discovery learning environment which helps students in overcoming these difficulties. We present design issues and the current state in the implementation of the system, with special focus on individualization of instruction towards a particular student.

1 Introduction

It is well known that one-on-one human tutoring is much more effective than traditional classroom instruction (Bloom 1984). The goal of the research in the area of Intelligent Tutoring Systems (ITS) is to build computerized tutors that achieve the effects of learning individually with a human tutor. It is a very ambitious goal, and currently available systems are nowhere near achieving it.

The research presented here has the same goal, with a focus on effective and computationally tractable ITSs. The system we work on is based on guided discovery learning and supports problem solving, conceptual and meta-learning. Our focus is on student models; they should be precise enough to guide instruction, and computationally tractable at the same time.

Here we present SQL-Tutor, an ITS for SQL programming. The system is currently being developed, with components in the various stages of implementation. This reports is intended to document the current state in the implementation of SQL-Tutor, as well as to inspire future work on this and similar systems. The report is structured as follows. We firstly discuss the choice of the domain for the system, and then expose the background of this research in section 3, by looking at the underlying learning theory, guided discovery learning, the field of ITSs in general and student modeling in particular. Section 4 is devoted to SQL-Tutor and deals with its architecture and components. The aspects of SQL-Tutor which have to be looked at are given in section 5. Finally, section 6 gives directions for future research and our plans for SQL-Tutor.

2 Why SQL?

SQL-Tutor has been developed for tutoring SQL, the dominant database language nowadays. SQL is the complete database language; it contains data and view definition statements, as well as data manipulation statements. It is a standardized language; the first standard appeared in 1986 and was updated in 1989, by the introduction of integrity enhancement features. SQL2 is the second version of the standard from 1992, the entry level of which is supported by most DBMS vendors today.

Although there is a movement towards graphical query interfaces, SQL is an extremely important development in the database world. It will be used for years to come either for interactive or programmed access to databases (embedded in other programming languages and tools for application development). SQL3, the latest standard scheduled to appear in 1998, is likely to gain even more importance for SQL, with the introduction of support for knowledge-based and OO applications and distributed databases, among other new features.

The author has been teaching SQL since 1991 as parts of Database Management courses at the University of Niš, Yugoslavia, and the University of Canterbury, New Zealand. The course is usually taken by about 50 students, and includes various topics, such as data models, database design, relational query languages (SQL, relational algebra and calculus), query processing, normalization theory, transaction processing and distributed databases. The practical part of the course consists of a database design assignment and a series of labs on an Ingres system.

Although SQL is a simple and highly structured language, students have lots of difficulties learning it. Some errors in students' queries come from the burden of having to memorize the database schema; the incorrect solutions may, for example, contain incorrect table or attribute names. Other errors come from misconceptions in student's understanding of the elements of SQL and the relational data model in general. Some of the concepts students find particularly difficult to grasp are grouping and restricting grouping. Join conditions and the difference between aggregate and scalar functions are another two common sources of confusion. Other

researchers report the same student misconceptions (Kearns et al. 1997).

SQL has been taught in the earlier mentioned courses, in the classroom, by solving problems on the blackboard, complemented by lab exercises. However, students find that is not easy to learn SQL directly by working with a DBMS, as the error messages are limited to the syntax only. Figure 1 illustrates a situation in which example 1 requires the student to specify a query with five clauses, as shown in the correct solution. When the student enters his/her incorrect solution, typically the error message generated by a RDBMS (Ingres in this case) will not be of much help. The same figure illustrates the kind of messages[1] the student may obtain from SQL-Tutor.

Example 1:

For each director, list the director's number and the total number of awards won by comedies he/she directed if that number is greater than 1.

Correct solution:	Student's solution:
SELECT DIRECTOR,SUM(AAWON)	SELECT DIRECTOR,SUM(AAWON)
FROM MOVIE	FROM DIRECTOR JOIN MOVIE
WHERE TYPE='comedy'	ON DIRECTOR=DIRECTOR.NUMBER
GROUP BY DIRECTOR	WHERE TYPE='comedy'
HAVING SUM(AAWON)>1	

INGRES:
E_USOB63 line 1, The columns in the SELECT clause must be contained in the GROUP BY clause.

SQL-Tutor:

- *You need to specify the GROUP BY clause!*
- *Specify the HAVING clause as well!"*
- *You do not need all the tables you specified!*
- *If there are aggregate functions in the SELECT clause, and the GROUP BY clause is empty, then SELECT must consists of aggregate functions only.*
- *For every table that appears in the FROM clause, there must be at least one attribute from that table used in any clause of the query.*

Figure 1: Inadequacy of feedback from a RDBMS

Figure 2 illustrates a situation of a semantic error. Instead of using the BORN attribute, the students specified the search condition on the DIED attribute, and the DBMS produced the result shown. The student may not even notice the error made. However, SQL-Tutor provides an appropriate message.

Many dialects of SQL exists, because the database vendors do not follow the standards. SQL-Tutor has been tailored to SQL as implemented by Ingres. The first version of the system, which is discussed in this paper, covers only the query part of SQL, that is the SELECT statement.

3 Background

There are several areas that provide the background for this research. Firstly, all ITSs are built on top of some learning theory; our beliefs in this direction are given in 3.1. As SQL-Tutor is

[1]Note that the student would usually be offered only one message at the time, as governed by the pedagogical rules. Here we show all possible messages for illustration.

Example 2:

List the names of all directors born in or after 1920.

Correct solution:	Student's solution:
SELECT LNAME, FNAME	SELECT LNAME, FNAME
FROM DIRECTOR	FROM DIRECTOR
WHERE BORN>=1920	WHERE DIED>=1920

Ingres:

lname	fname
Hitchcock	Alfred
De Mille	Cecil
Ford	John

SQL-TUTOR:

- *Check that you are comparing the numerical constant to the right attribute in the WHERE clause!*

Figure 2: Inability of a RDBMS to deal with semantic errors

designed as a guided discovery learning environment, section 3.2 explores this teaching strategy. Section 3.3 briefly discusses the history of the ITS field and focuses on the typical architecture of such systems. Student modeling (SM) is the topic of section 3.4. The intractable nature of SM is briefly discussed, and the majority of the section is devoted to the SM approach SQL-TUTOR is built around.

3.1 Learning theory

In our view, existing learning theories touch upon various aspects of learning, but there is no single theory that covers all of them. It is not the goal of our research to develop a learning theory. However, we do have a view on learning which is the integration of ideas from several theories of learning: Anderson's ACT^* and $ACT-R$ theories (1983, 1993), VanLehn's Impasse-Driven Learning (1988), the general architecture of intelligence SOAR (Rosenbloom et al. 1986) and Ohlsson's theory on learning from performance errors (1996).

We believe that learning goes in three phases. The first, relatively short phase, consists of acquisition of new knowledge in declarative form, and is also referred to as conceptual learning (VanLehn 1996). In this phase, students learn in lectures and by reading books.

The second phase is much longer, and there are several processes which take place in it. VanLehn (1996) mentions two processes: practising the application of existing knowledge and acquiring new knowledge. The former consists of problem solving, i.e. applying new knowledge acquired in the first phase. This process also appears in Anderson's theories, when knowledge is transformed from a declarative into the procedural form. Procedural knowledge is more specific, because it focuses on specific aspects of problems, and therefore easier to apply.

VanLehn believes that the second process is used by students when they reach impasses and realize that they miss some parts of domain knowledge. A student tries to get the missing knowledge from any source available - teachers, books or peers. However, in test situations, where such help is not available, students engage in error repairs (Brown & VanLehn 1980). Impasses require the student to perform local problem solving strategies. The student knows the starting state (being stuck) and the final state (being unstuck), and then applies operators that do not change the state of the problem, but the state of the rule interpreter itself (VanLehn 1988). Bugs are the result of unsuccessful attempts to extend the existing knowledge to new

situations.

Another important process not considered by VanLehn is learning from errors. Errors are the results of overly general knowledge. Ohlsson (1996) explains the process of learning from errors as consisting of two parts: error recognition and error correction. Declarative knowledge is required for an error to be detected and then the error can be corrected by specializing faulty knowledge so that it is applicable only in situations in which it is appropriate.

The third phase happens only occasionally; it transforms a student into an expert by several processes. One of them is called knowledge compilation by Anderson, and is very similar to the process of chunking in SOAR. Basically, the student becomes proficient in using the knowledge (strengthening with practice) and combines several primitive operators into a more complex operator. This is also known as success-driven optimization of existing knowledge. In this phase students are also engaged in meta-learning, i.e. acquisition of new strategies of learning which are more effective than those used.

3.2 Guided discovery learning

Recently there have been many systems based on guided discovery as the teaching strategy. There are psychological studies (Anderson 1993) which show that students learn better from discovery than from direct instruction and that such knowledge is retained for longer than when learning by being told. Student memorize the things they have discovered themselves much better. Of course, unrestricted exploration is not advisable, especially for novices, because students may waste too much time wondering. The solution is found in providing guidance, in form of solicited on unsolicited hints from the tutor.

SQL-Tutor is designed to be a problem solving environment for learning-by-doing. It is not intended to replace classroom instruction, but to complement it; therefore, we start from the assumption that the student is already familiar with the database theory and fundamentals of SQL. Students work on their own as much as possible and the system intervenes when the student is stuck or asks for help. In such a way, students maintain a feeling of control.

3.3 Intelligent Tutoring Systems

One of the reasons for the crisis in education is the large number of students before which the instructions are carried. Since individualized instructions are impossible, the content and timing of the knowledge transfer process have to be tailored to the needs and abilities of a group of students.

A way to overcome this crisis is the development of computer–aided educational systems. The use of computers in education began in 1950's with CAI (Computer-Aided Instruction) systems, which were statical programs designed for specific areas. First CAI systems were completely specified during their design and left no way to choose the order or content of instructions. In 1960s a new generation of CAI systems appeared, often referred to as *branching programs*, which used student's response to control the material he/she was shown. Each question a CAI system could pose to a student was determined in advance, as well as several possible answers to it. Branching was based on comparison of student's answer to these prespecified answers. Such an approach could result in classifying student's answers as incorrect if they were given in different forms from those predicted by the designer. The main disadvantages of CAI systems were the low level of individualization offered to students and the poorness of feedback, both caused by their knowledge-sparse nature.

First attempts to make CAI programs "intelligent" started in 1970s. In this paper we use the term Intelligent Tutoring Systems (ITS) for what have been called Intelligent Computer-Assisted Instruction (ICAI), Intelligent Educational Systems (IES), knowledge-based tutoring

systems and Intelligent Learning Environments (ILE). All of these terms describe knowledge-based systems which shift the focus from knowledge transfer to knowledge communication. The latter subsumes a student actively participating in the learning process adapted to his/her knowledge and learning capabilities. ITSs provide individualized instructions by encapsulating pedagogical, domain and communication knowledge with the knowledge about the student. However, there is no single line between CAI and ITS systems – real systems are points on a spectrum between fixed, preprogrammed decisions and knowledge-based adaptive behaviour.

The area of ITS is an interdisciplinary field, covering the diversity of interests of artificial intelligence, psychology, education, antrophology, linguistics, cognitive science and man-machine interaction. Although ITS research has been going on for more than two decades now, only a few systems have been in regular use in classrooms, due to complexity of the task they are to perform and small, but increasing number of the researchers. The discipline is undergoing a transformation from a pure research stage to a developmental phase where it may start to have practical significance.

Typical architecture of an ITS includes four components, as shown in figure 3. A *domain module* (also called expert module) contains domain knowledge needed for generation and/or solving the problems to be given to the student, as well as for student diagnosis. *Student modeler* is the component which generates student models, which describe students' knowledge and learning abilities, as detected by the system through the dialogue. *Pedagogical module* determines the way of communication, i.e. the timing, style and content of tutor's interventions. An *interface* serves as a mediator between the student and the system and should provide enough information about the system and be robust, flexible and easy to use.

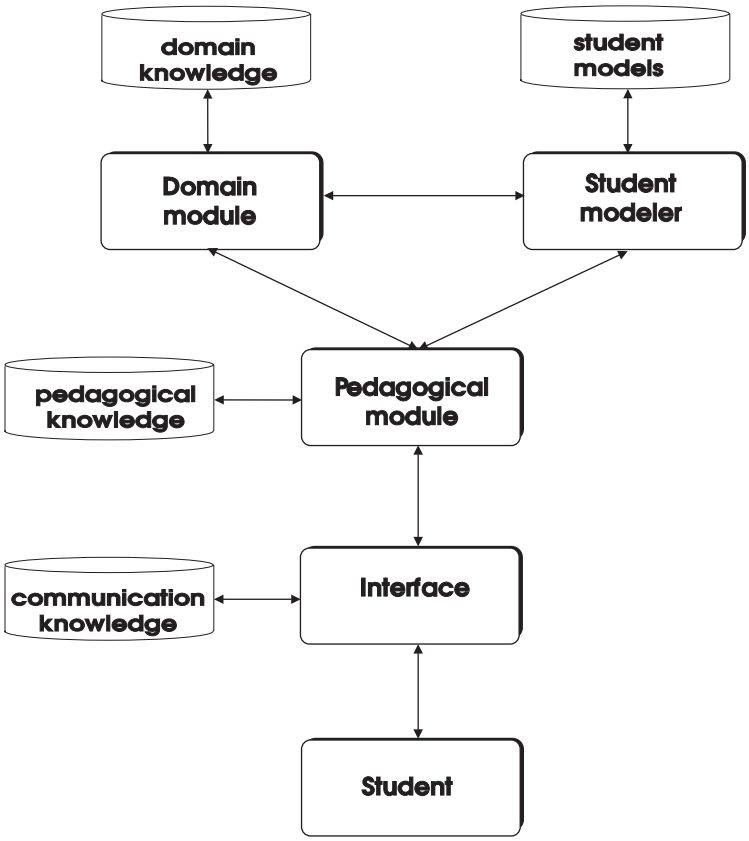

Figure 3: Typical architecture of ITSs

3.4 Student modeling and CBM

The effectiveness of ITS systems depends critically on their ability to offer individualized instruction. In other words, they should be able to adapt to the knowledge, learning capabilities and general characteristics of their users. In doing so, these systems employ student models, which enable the selection of instructional content and tutorial strategy, as well as diagnoses confirmation. The teacher, be it a natural or an artificial one, has to understand the student in order for intelligent communication to happen. Student modeling (known also as student diagnosis) can be defined as the process of gathering relevant information in order to identify and represent the knowledge state of the student.

In an ideal case the model of a student should illustrate his/her knowledge, learning strategies preferred, areas of interest besides that of instruction, preferred presentation style, level of concentration and so on. There is little consensus among the researchers in the area of intelligent education about the terminology used or methodology for student modeling. Several techniques for student modeling have been developed for particular domains, the generality of which is yet to be determined by applying them elsewhere. The survey of SM approaches is outside the scope of such a paper and the interested reader is referred to (Wenger 1987, Greer et al. 1994).

A student model is always an approximation of his or her knowledge and state of mind due to noise present during the formation of the model. Noise comes from several sources, such as inconsistent student behaviour, caused by the loss of concentration or tiredness, dynamic and non–monotonic nature of human learning, ambiguities and indeterminacy in student's answers, caused by limitations of the communication channel.

The task of building a student model is extremely difficult and laborious, due to huge search spaces involved. Several researchers have pointed to the inherent intractability of the task (Self 1990; Ohlsson 1994; Holt et al. 1994). Various approaches to dealing with the intractability of student modeling have been introduced. Self (1990, 1994) recommends such design of the interactions that information necessary for building a student model is provided by the student, and not inferred by the system. Also, it is not useful to be able to identify misconceptions in the student knowledge which cannot be dealt with by the tutor. An ITS should model only what it is capable of using in order to generate remedial or other pedagogical actions.

For a number of years we have been uneasy about the requirement for cognitive validity of student models. If the goal is to model student's knowledge completely and precisely, student modeling is bound to be intractable. However, a student model can be useful even if it is not complete and accurate (Self 1994; Woolf & Murray 1994; Ohlsson 1994). Even simple and constrained modeling is sufficient for instruction purposes, and this claim is supported by findings that human teachers also use very loose models of their learners, and yet are highly effective in what they do (Holt et al. 1994; Self 1994). Anderson (1993) also argues against such modeling, saying that it is not very useful for students to be told about the sources of their misconceptions, and that students benefit much more by being given an informative error message. There are other student modeling approaches that do not insist on detailed student models (Stern et al. 1996, Beck et al. 1997).

One of the SM approaches that focus on reducing the complexity of the task is Constraint-Based Modeling (CBM) (Ohlsson 1994). CBM is based on Ohlsson's theory on learning from performance errors (Ohlsson 1996), discussed in 3.1. Ohlsson's approach focuses on faulty knowledge, realizing that it is not sufficient to describe what the student knows correctly. The basic assumption is that diagnostic information is not hidden in the sequence of student's actions, but in the situation (or the problem state) that the student arrived at. This assumption is supported by the fact that there can be no correct solution of a problem that traverses a problem state which violates the fundamental ideas or concepts of the domain. The student model does not represent student's actions, but the effects of his or her actions instead.

Because the space of false knowledge is vast, much more so than the space of correct knowledge, Ohlsson suggest the use of an abstraction mechanism which he realizes in the form of state constraints. A state constraint is an ordered pair $(C_r,\ C_s)$, where C_r is the relevance condition and C_s is the satisfaction condition. C_r is used to identify the equivalence class, or the class of problem states in which C_r is relevant. C_s identifies the class of relevant states in which C_s is satisfied. Each constraint specifies the property of the domain which is shared by all correct paths. In other words, if C_r is satisfied in a problem state, in order for that problem state to be a correct one, it must also satisfy C_s. Conditions may be any kinds of logical formulas, hence may be constructed on various tests on the problem state in question. An example of a constraint in the arithmetic domain is (Ohlsson 1994):

If the problem is $n_1/d_1 + n_2/d_2$ *and if* $n = n_1 + n_2$,
then it had better be the case that $d_1 = d_2$.

The relevance conditions focuses on a class of problems in which two fractions are to be added and the numerator of the result equals to the sum of the numerators of the starting fractions. In all such problems, the solution can only be correct if the denominators of the starting fractions are the same. In other words, there is no correct sum of the starting fractions that has a numerator equal to the sum of numerators unless the denominators are identical.

In such a way, CBM represents domain knowledge as a set of state constraints. Constraints define sets of equivalent problem states. An equivalence class triggers the same instructional action; hence the states in an equivalence class are pedagogically equivalent. All correct problem solutions do not violate any of the constraints. A violated constraint signals the error, which translates to incomplete and incorrect knowledge.

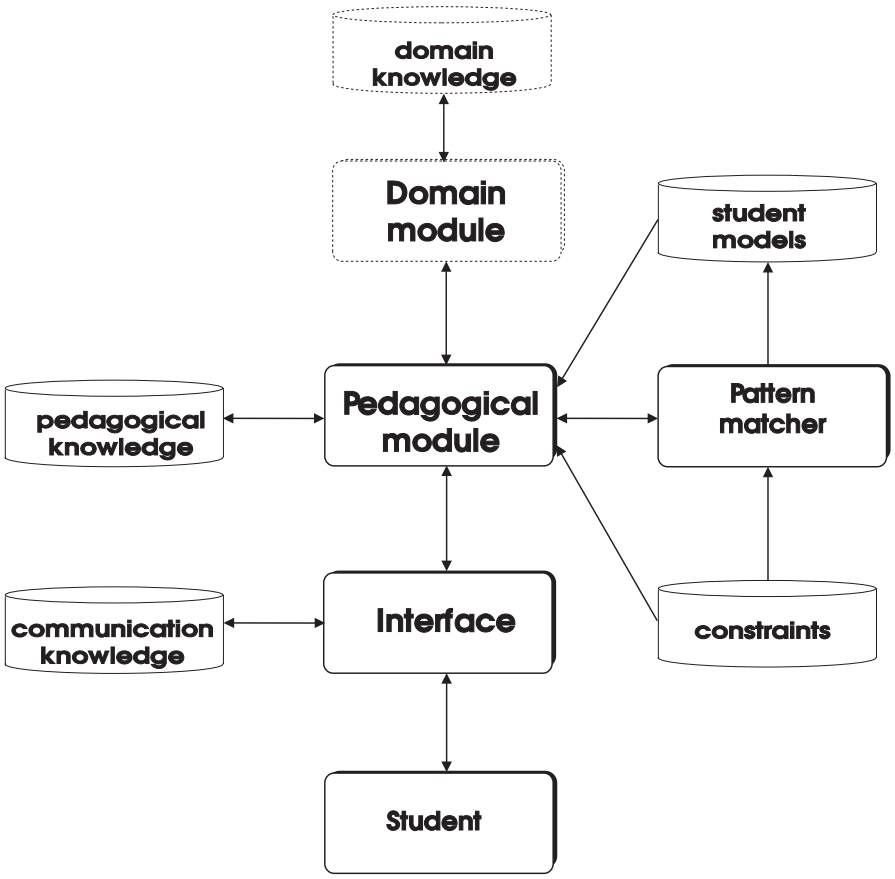

Figure 4: Architecture of CBM-based ITSs

The architecture of ITSs based on CBM (figure 4) would therefore be simpler than the typical one, shown in figure 3. One of the advantages of CBM lies in its non-insistence on a runnable expert module; in many domains it can be very difficult to build one. CBM-based computerized tutors are able to generate instructional actions even without being able to solve problems on their own, by focusing on violated constraints. Of course, CBM does not prevent an ITS from having a domain module. On the contrary, the existence of a domain module can be very beneficial to the student, as it can provide the answer to student's questions such as "What do I do next?". Expert modules are based on a different type of knowledge (i.e., prescriptive) and hence can be used for such purposes.

Another advantage of CBM is its computational simplicity. Instead of using complex reasoning as required by other diagnostic approaches, CBM reduces student modeling to pattern matching. Conditions are combinations of patters, and can therefore be represented in compiled forms, such as RETE networks (Forgy 1982), which are very fast and and for which off-the-shelf software is available. In the first step all relevance patterns are matched against the problem state. In the second step, the satisfaction components of constraints that matched the problem state in the first step (i.e., relevant constraints) are matched. If a satisfaction pattern matches the state, then the constraint is satisfied, and the ITS is not to take any action. In the opposite case, the constraint is violated. The student model consists of all violated constraints. This technique can be used for both on– and off–line student modeling

Furthermore, CBM does not require extensive studies of students bugs as in enumerative modeling (Anderson & Jeffries 1985). Another deficiency of many SM approaches that CBM is not sensitive to is the *radical strategy variability phenomenon.* Namely, some approaches assume that a student systematically uses a single procedure for the task at hand. However, Ohlsson shows (Ohlsson 1994) that each student applies just one of a family of procedures applicable to the problem at hand, and that the procedure selection strategy is ad hoc. Ohlsson claims that each student knows several procedures at the same time and that he or she can switch between them on different problems. On the other hand, CBM allows students to be inconsistent; it ignores the particular strategy the student applied, since different strategies may result in the same constraint broken. The approach is also neutral with respect to the pedagogy, since different pedagogical actions (immediate or delayed ones) may be generated on the basis of the model.

So far, there have not been any ITSs based on CBM. Ohlsson claims that CBM may prove to be superior to other approaches, because of the advantages discussed. However, there are some potential disadvantages that need to be checked for. It is not clear whether constraints provide the right type of abstraction for representing knowledge in various domains. It may also be true that constraints provide too loose a net so that some of the students inconsistencies may not be identified by the tutor. One of the goals of this research is to test CBM in reality. The next section present the design and implementation issues of SQL-Tutor, an ITS based on CBM.

4 SQL-Tutor

SQL-Tutor has been developed for tutoring SQL and is aimed at upper-level undergraduate students. The system is implemented in CLOS (Franz 1996) on SUN workstations and will soon be ported to PC compatibles. Definitions of all the SQL-Tutor classes are given in Appendix 1.

SQL-Tutor is designed as a practice environment; we suppose that students have previously been exposed to the concepts of database management in lectures. Therefore, the system is not a substitute for the conventional style of education, but a complement to it. The system currently covers only the SELECT statement of SQL, but the same approach could be used with other

SQL statements. This focus on the SELECT statement does not reduce the importance of the system, because queries cause the most misconceptions for students. Moreover, many of the concepts covered by SELECT are directly relevant to other SQL statements and other relational database languages in general.

4.1 Architecture of SQL-Tutor

As illustrated in figure 5, SQL-Tutor has a very simple architecture. Domain knowledge is represented in the form of constraints, as discussed in section 4.3. The system contains definitions of several databases, which are also implemented on the RDBMS used in the lab (currently Ingres). New databases can easily be added to SQL-Tutor, by supplying the same SQL files used to create the database in the RDBMS. Appendix 2 contains the files used to define the databases.

SQL-Tutor also contains a set of problems for specified databases and the ideal solutions to them. The solutions are necessary because SQL-Tutor has no domain module and is not capable of solving problems. The rationale for such a departure from the typical architecture of an ITS, which also includes a domain module, follows. Designing an ITS to teach SQL presents various difficulties. Database queries are given in a natural language; however, the current state-of-the-art in Natural Language Processing (NLP) is still far from being able of handling various problems present in such queries, such as references and synonyms. There is a possibility to circumscribe the NLP problem: the text of the problem may be represented not in its natural-language form, but in a form which could be the product of NLP, as done in (Anderson et al. 1995). However, it is hard not to build parts of the solution into such a representation. Furthermore, even if we overlook the NLP problem, the knowledge required to write SQL queries is very fuzzy. Therefore, it would be very difficult, if not entirely impossible, to develop a problem solver in this area.

In spite of the difficulties with problem-solving, an ITS needs a way to evaluate student queries. SQL-Tutor does this by comparing student solutions to ideal ones. That is the reason for SQL-Tutor to require ideal solutions to problems.

The basic components of the system are the interface, pedagogical module and the CBM-based student modeler. The pedagogical module (PM) observes every student's action performed in the interface, and reacts to it appropriately. At the beginning of the interaction, a problem must be selected for the student to work on. When the student enters the solution for the current problem, PM sends it to the student modeler, which propagates the solution through the relevance network first, and then through the satisfaction network for relevant constraints. All the violated constraints are identified and the student model is updated accordingly, as discussed in section 4.4. PM then generates an appropriate pedagogical action (e.g. feedback). When the current problem is solved, or the student requires a new problem to work on, the pedagogical module selects an appropriate problem on the basis of the student model. The following subsections discuss the individual components in more detail.

4.2 Interface

The interface of SQL-Tutor, illustrated in figure 6, has been designed with several pedagogical and HCI guidelines in mind. Generally, interfaces for ITSs should be robust, flexible, easy to use and understand. An interface is a mediating device; hence it must provide information about the system itself. At the same time, ITS interfaces are problem–solving environments and therefore should be similar to real environments and support reification of the goal structure. Furthermore, as learning is a very difficult task, the interfaces should reduce the working-memory load of students. Here is how we handled the above criteria.

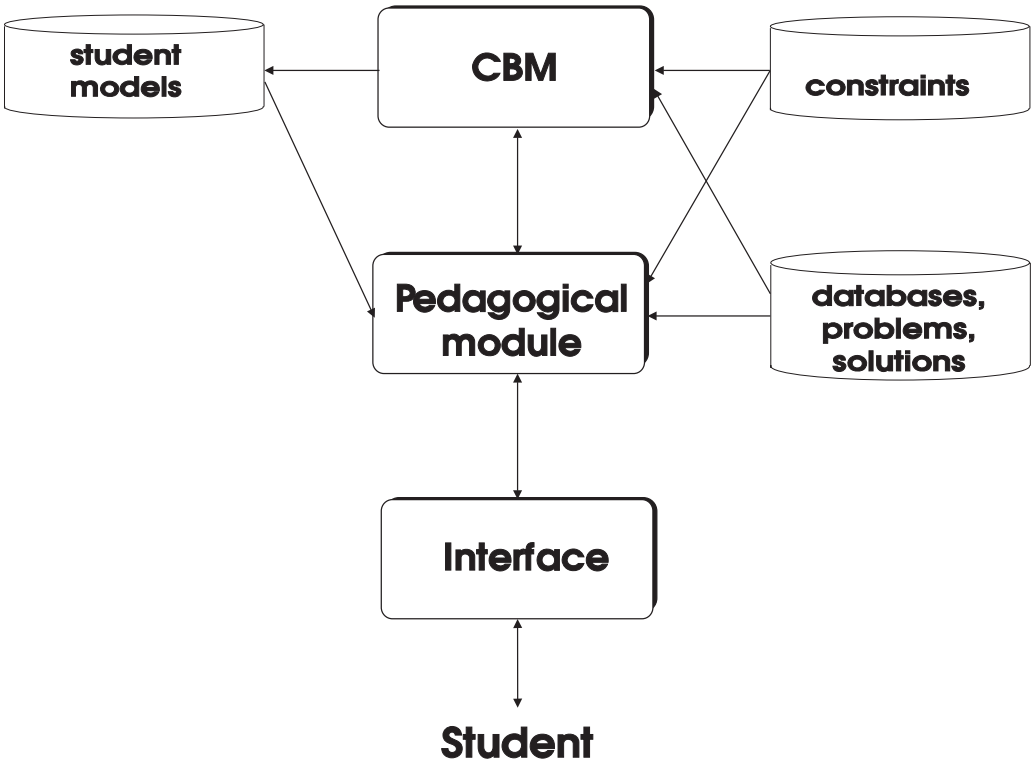

Figure 5: Architecture of SQL-Tutor

The interface of SQL-Tutor supports problem-solving in several ways. It reduces the memory load by displaying the database schema and problem text, providing the basic structure of the query and also by providing explanations about the elements of SQL. The main window of SQL-Tutor is divided into three areas which are always visible to the student. The upper part of the window displays the text of the problem being solved and the student can always remind him/herself easily of the elements requested in the query. The middle part contains the clauses of the SQL SELECT statement, thus visualizing the goal structure. Students need not remember the exact keywords used and the relative order of clauses. The lowest part displays the schema of the currently chosen database. The schema name is given first, followed by the descriptions of tables. Each table is shown by its name and schema enclosed in a box. The name(s) of the attribute(s) forming the primary key is underlined and given in blue. The foreign key attributes are given in red.

The visualization of a schema is quite important; all database users are painfully aware of the constant need to remember table and attribute names and the corresponding semantics as well. SQL-Tutor users can ask for the description of databases, tables or attributes by selecting the appropriate options from the Help menu, or by directly selecting table/attribute names (figure 7). Furthermore, the users can ask for descriptions of any elements of SQL, such as functions, expressions, predicates, operators and others, by selecting appropriate options in the Help menu (figure 8). The motivation here is to remove from the student some of the cognitive load required for checking the low-level syntax and to enable the student to focus on higher-level query definition problems.

The interface also provides a description of the whole system in order to enable the student to form an overall understanding. This is implemented in the interface of SQL-Tutor via the About SQL-Tutor option in the Help menu (figure 9).

In SQL-Tutor, the reification of the goal structure is supported by visualizing the elements (clauses) of an SQL query. The student can obtain short description of the roles of various

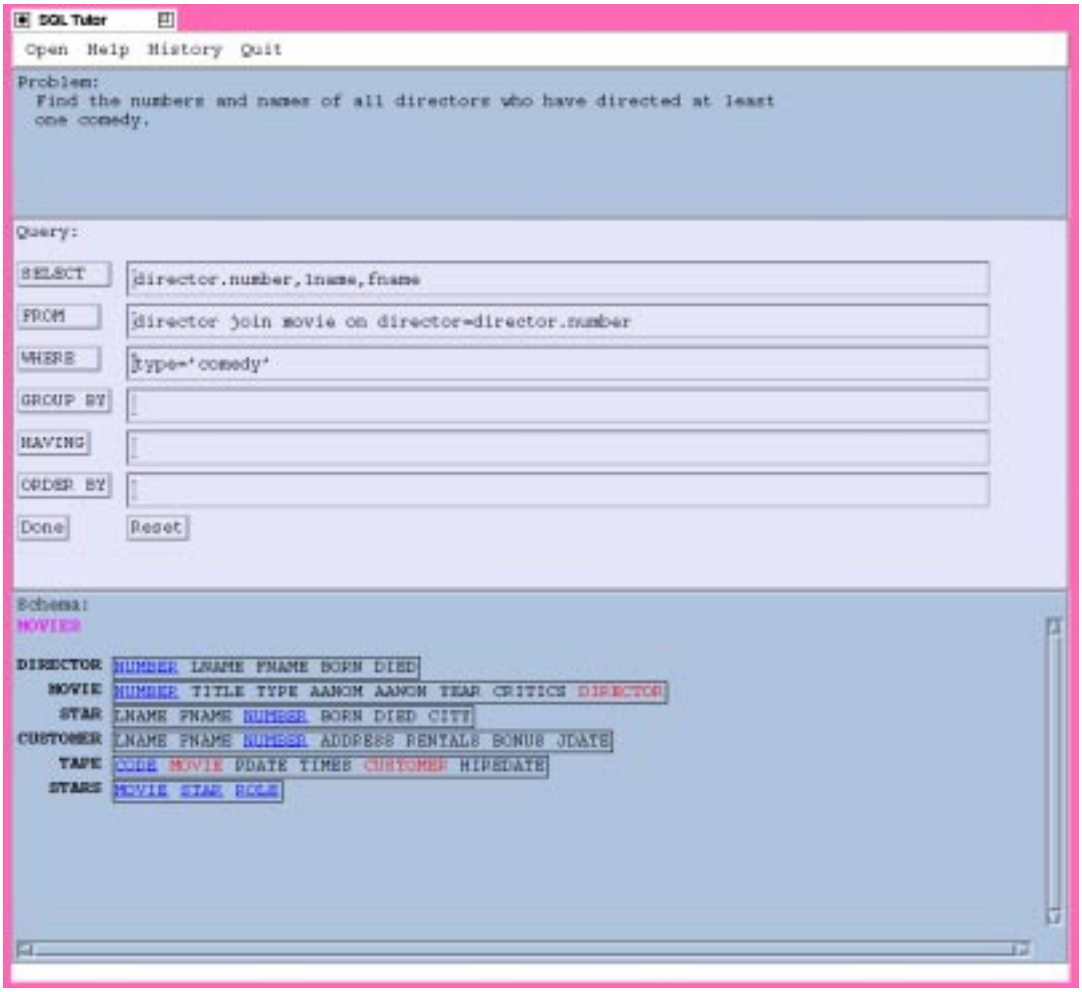

Figure 6: The interface of SQL-Tutor

clauses by selecting the appropriate clause or by asking for help from the main menu. This also illustrates the use of a CHI concept known as equal opportunity controls (Thimbleby 1990). This concept provides several opportunities to the user for representing his/her request to the system. Sometimes, the user will select an option from the menu, or perform a direct manipulation action, or enter the request textually or in some other form. There are other elements of SQL-Tutor's interface which are based on the same concept.

4.3 Constraints

The process of identifying constrains is the knowledge acquisition process, with all the known problems associated with it. SQL-Tutor models students by looking at the student's solution and by comparing the student's solution to the ideal one. Because of this decision, constraints are divided into two groups; one dealing with the syntax of SQL and the other dealing with semantics of queries, by focusing on the differences between the student's and the expert's solution.

The constraint base of SQL-Tutor currently consists of 199 constraints, which are acquired by analyzing the domain knowledge (Elmasri & Navathe 1994; Pratt 1990) and on the basis of a comparative analysis of correct and incorrect solutions. It is well known that knowledge acquisition is a very slow, time-consuming and labour-intensive process. Anderson (1995) reports 10 or more hours necessary for induction of a production rule. When interviewing domain experts

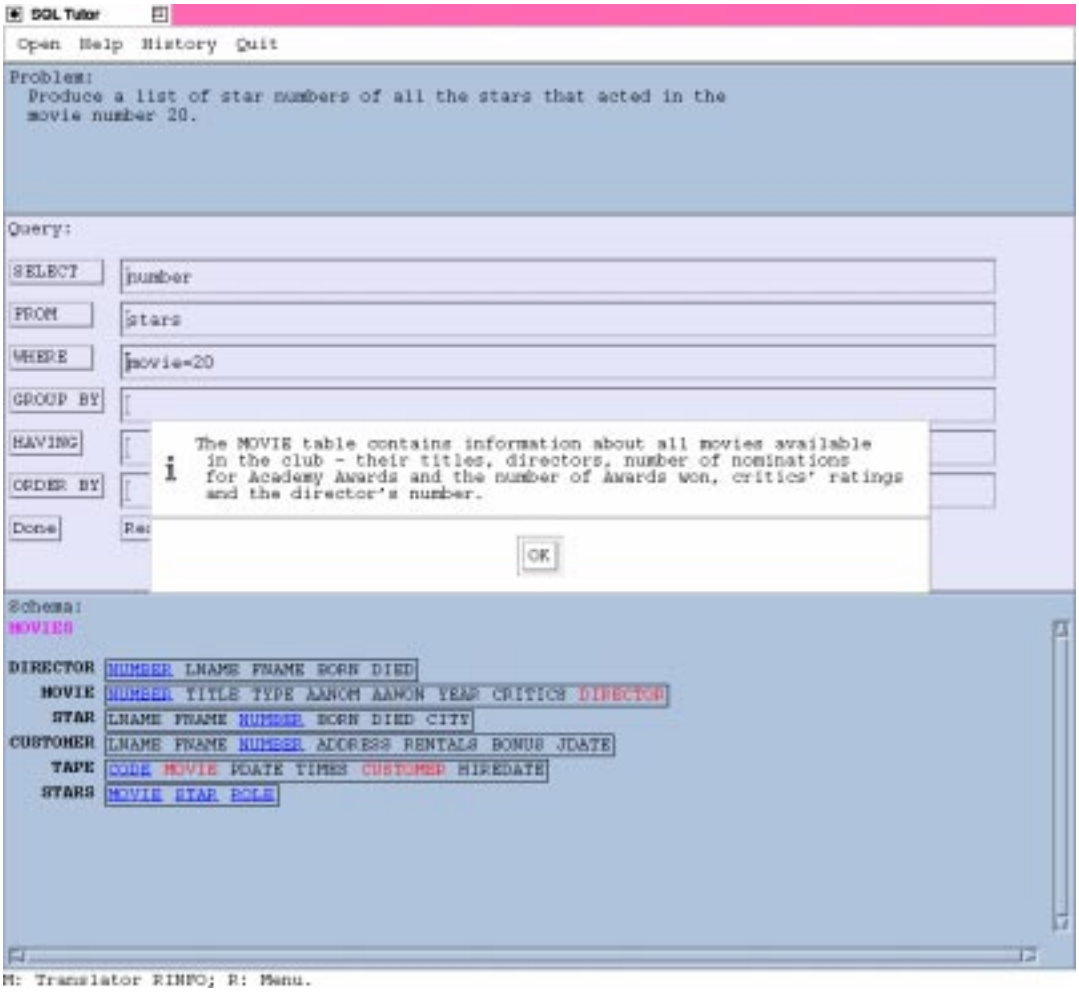

Figure 7: Describing tables in SQL-Tutor

in order to acquire knowledge for expert systems, usually 2 to 5 production rules equivalents are identified per day. The time spent on identification, implementation and testing of SQL-Tutor's constraints is 1.3 hours per production, which is significantly shorter than those above. This may be the consequence of the same person serving as the domain expert and knowledge engineer (and the system developer, at that matter), but may also illustrate the appropriateness of the chosen formalism.

We discuss constraints in more detail next, turn to pattern definition in 4.3.2, give several examples of constraints in 4.3.3. and look at how they are compiled to speed up the pattern matching process in 4.3.4.

4.3.1 Defining constraints

The definition of the `constraint` class is given below. After specifying the constraint's number (unique), the next two slots contain the relevance and satisfaction patterns. The *problems* slot contains a list of problem numbers for which the constraint is relevant. The next slot contains the description of the constraint, an error message that is displayed to the student as the hint in case of violation of the constraint. The last slot, (*clause*), is again used for generating feedback to the student; it specifies the clause of the SELECT statement to which the constraint applies and is used to generate the error-flag type feedback, as explained in 4.5.

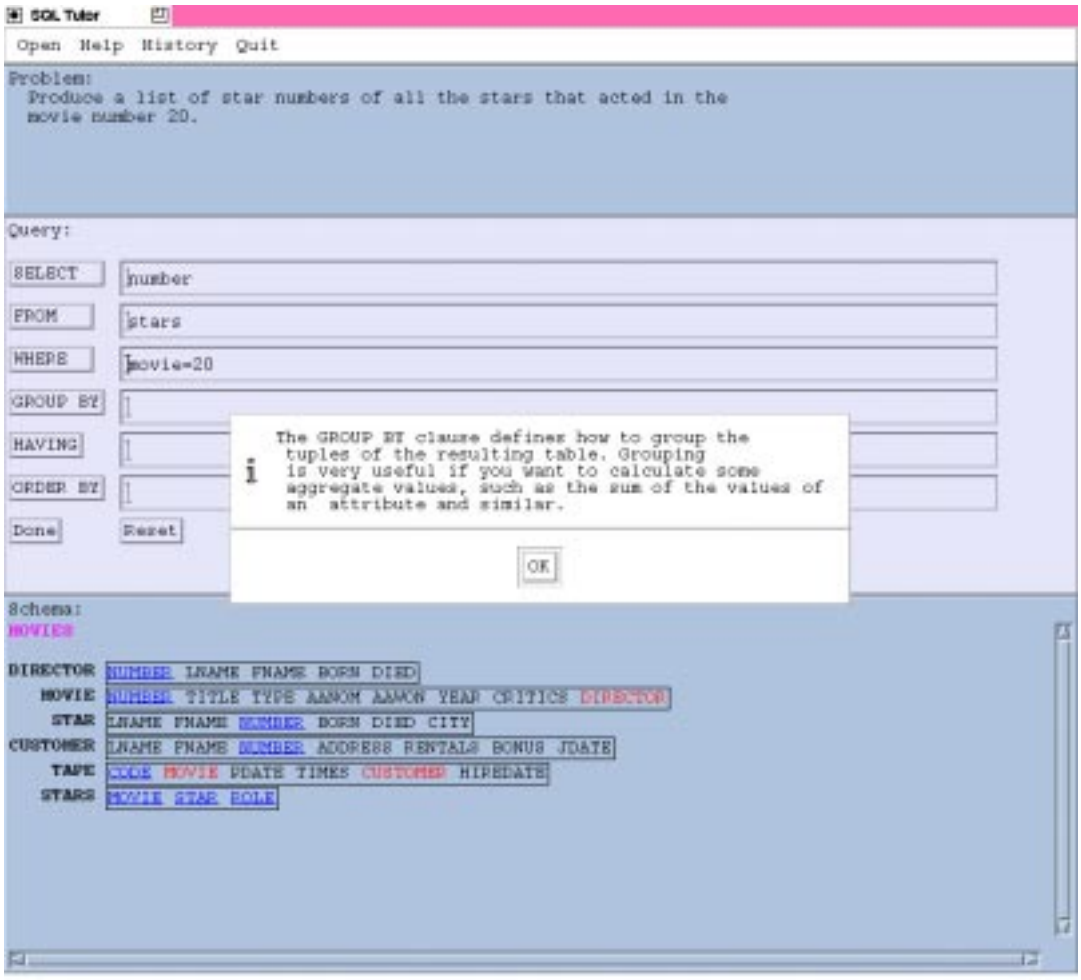

Figure 8: Conceptual help in SQL-Tutor

```
(defclass     constraint  ()
((number      :type       integer   :accessor  number       :initarg  number)
(rel-pattern  :type       cons      :accessor  rel-pattern  :initarg  rel-pattern)
(sat-pattern  :type       cons      :accessor  sat-pattern  :initarg  sat-pattern)
(problems     :type       cons      :accessor  problems     :initarg  problems)
(descr        :type       string    :accessor  descr        :initarg  descr)
(clause       :type       string    :accessor  clause       :initarg  clause)))
```

Relevance and satisfaction patterns can be any logical formulas, consist of any number of conditions. Some of the conditions match parts of the student's solution to prespecified patters or the ideal solution. Other conditions may be any Lisp functions.

4.3.2 Specifying patterns

Pattern matching is a well known technique in which a pattern (i.e., an expression containing variables) is compared to a constant expression (with no variables) to see whether they are similar. Patterns contain variables, that can represent any kind of expressions, and constants, which must appear in the constant expression as they are given in the pattern. Pattern matching tries to find substitutions for the variables in the pattern that will make the pattern identical to the constant expression.

In SQL-Tutor, very often a part of the student's solution is compared to a prespecified pattern. Patterns are specified by using the following symbols (Norvig 1992):

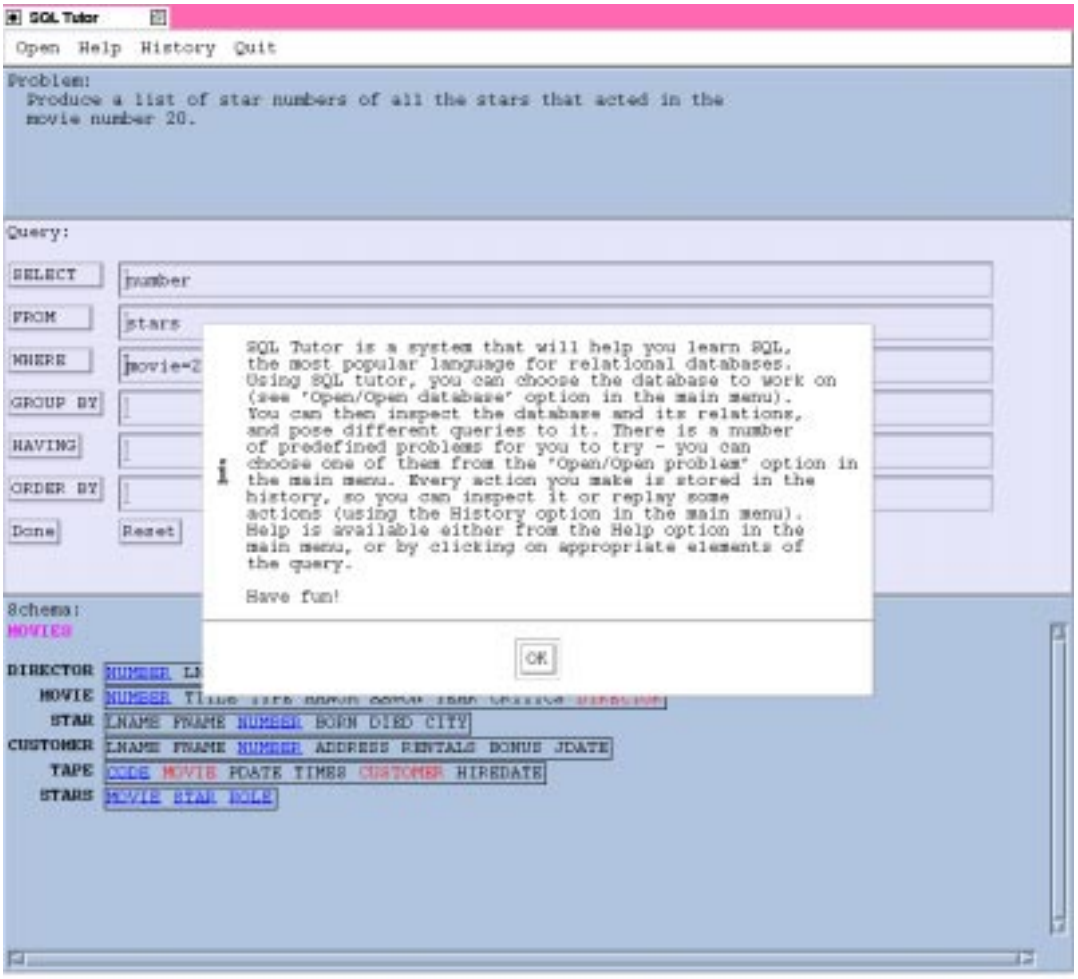

Figure 9: Description of SQL-Tutor

`?var`	A simple variable - matches any one expression
`?*var`	Matches zero or more expressions
`?+var`	Matches one or more expressions
`??var`	Matches zero or one expression
`(?if exp)`	Tests if `exp` is true
`(?is var predicate)`	Test predicate on one expression
`(?or pat ...)`	Match any pattern on one expression
`(?and pat ...)`	Match every pattern on one expression
`(?not pat ...)`	Succeed if pattern(s) do not match

We discuss the usage of these symbols on several examples. For instance, if we want to check whether the student uses the IS NULL predicate correctly, we can specify the following pattern:

'(?*d1 ?a "IS" ??p "NULL" ?*d2)

This pattern will match any list of strings[2] that contains zero or more strings at the beginning, a string followed by the IS keyword, possibly another string and the NULL keyword, and finally zero or more strings at the end. The pattern will match the following lists of strings:

[2]The student's solution is stored as 6 lists of strings, one for each clause of the SELECT statement.

'("DIED" "IS" "NULL")[3]
'("DIRECTOR" "=" DIRECTOR.NUMBER" "AND" "DIED" "IS" "NULL")
'("SUPERVISOR" "IS" "NOT" "NULL")[4]

However, the same pattern would not match the following lists:

'("DIED" "=" "NULL")[5]
'("DIED" "<>" "NULL")[6]

Let us turn to another example now: we want to check whether there is a condition that checks whether an expression is greater than a numeric constant. This situation can also be specified in a pattern that can be matched to a student's query. The pattern is:

'(?*d1 ?n ">" (?is ?c1 numericp) ?*d2)

We allow any number of strings at the beginning and the end of the list the pattern is being matched against, in order to be as general as possible[7]. Note that the pattern also checks whether `?c1` is a numeric constant, by using the `?is` symbol. The pattern matches the following WHERE clause of a student's query:

'("AAWON" ">" "0" "AND" "YEAR" ">" "1987")[8]

In another situation, we may want to check whether the student uses the BETWEEN predicate properly. This predicates tests whether the value of an attribute belongs to a given interval, specified by its lower and upper value. The proper usage of BETWEEN can be specified with the following pattern:[9]

'(?*d1 (?is ?a attribute-p) ??n "BETWEEN" ?v1 ??p ?v2 ?*d2),

which matches the following:

'("YEAR" "BETWEEN" "1992" "AND" "1993")
'("YEAR" "BETWEEN" "1992" "," "1993")
'("YEAR" "BETWEEN" "1992" "1993")

The last two constant expressions are incorrect. Constraint 22[10] contains additional conditions to check whether the `??p` variable is bound or not, and if it is bound, that it is bound to "AND".

In a more complex situation, we want to see whether a student knows how to specify embedded SELECT statements in comparisons. An embedded SELECT requires an attribute to be compared to and various comparison operators may be used to define the condition (<, >, =, ! =, <>, <=, >=). The inner (i.e. embedded) SELECT must be specified within brackets. Such comparison may be the only condition in the WHERE clause, but other conditions are

[3]The `?a` variable is bound to the DIED attribute.
[4]The `?a` variable is bound to the SUPERVISOR attribute, and `??p` is bound to NOT.
[5]Here a student has used = instead of the IS keyword.
[6]Here a student has used <> instead of IS NOT.
[7]Such a pattern will allow the condition to appear at the beginning, in the middle or at the end of the list.
[8]There are two ways for this pattern to match the list: in the first case, `?n` is bound to AAWON, and in the second, it is bound to YEAR.
[9]The *attribute-p* predicate checks whether an expression is a valid attribute in the current database.
[10]This constraint is given in 4.3.3.

also allowed before or after it. The pattern to be used in this situation is:

```
'(?*d1 ?a (?or "<" ">" "=" "! =" "<>" "<=" ">=")
          "(" "SELECT" ?+la "FROM" ?+r ")" ?*d2)
```

Note that the SELECT clause of the embedded query may contain one or more expressions, and that at least the FROM clause must be specified[11].

4.3.3 Examples of constraints

The constraints differ significantly. An example of a very simple constraint is given below, illustrating a constraint which is applicable to all SELECT statements. The relevance pattern of this constraint is 't, which is always satisfied; therefore, this constraint is relevant to all student's solutions. The satisfaction pattern specifies that the SELECT clause cannot be empty.

```
(p 2
"The SELECT clause is a mandatory one.
Specify the attributes/expressions to retrieve from the database."
t
(not (null (slot-value ss 'select)))
"SELECT")
```

However, constraints can be much more complex. We have seen how patterns can be used for checking for specific situations. As said earlier, other conditions may be any LISP functions, and they can use the binding lists obtained from the matching process to perform additional tests on the student's query. Constraint 186, given below, consists of very complex conditions. It applies to situations where the WHERE clause of the ideal solution contains (at least one) condition which checks whether the value of a numeric attribute is greater than some numeric constant `c1` and the same attribute appears in the student's solution in a condition with greater-than-or-equal instead of the grater-than operator. If that is the case, the satisfaction pattern ensures that the constant in the student's solution is 1 less than the constant in the ideal solution. The relevance condition of this constraint contains a pattern (which has already been discussed) which is matched against the WHERE clause of the student's solution. The constraint also searches for a similar pattern in the ideal solution of the problem.

```
(p 186
"Check the numerical constant you are using in the WHERE clause!"
(and (not (null (where ss)))
     (not (null (where is)))
     (bind-all '?n (find-names ss 'where) bindings)
     (attribute-in-db (find-schema (current-database *radni*)) '?n)
     (equalp (find-type '?n) 'numeric)
     (match '(?*d1 ?n ">" (?is ?c1 numericp) ?*d2) (where ss) bindings)
     (match '(?*d3 ?n ">=" (?is ?c2 numericp) ?*d4) (where is) bindings))
(equalp (string-to-number '?c2) (1+ (string-to-number '?c1)))
"WHERE")
```

The effect of constraint 67, given below, is illustrated in figure 1. If the GROUP BY clause of the student's solution is empty, and the SELECT clause contains at least one aggregate function, the satisfaction condition of this constraint specifies that the only expressions allowed in the SELECT clause are aggregate functions. In figure 1, the student asked for the values

[11] If the embedded query contain other clauses as well, they will be bound to the `?+r` variable.

of the DIRECTOR attribute to be shown as well, and that is not possible as RDBMS does not know how to group the tuples of the table (i.e. there are many values of the DIRECTOR attribute).

```
(p 67
"If there are aggregate functions in the SELECT clause,
and the GROUP BY clause is empty,
then the SELECT clause must consists of aggregate functions only."
(and (not (null (intersection '("MIN" "MAX" "COUNT" "AVG" "SUM")
                  (slot-value ss 'select) :test 'equal)))
     (null (slot-value ss 'group-by))
     (bind-all '?e (get-expressions (slot-value ss 'select)) bindings))
(not (null (intersection '("MIN" "MAX" "COUNT" "AVG" "SUM") '?e :test 'equal)))
"SELECT")
```

Constraint 22 checks whether students use the BETWEEN predicate correctly. Its relevance condition makes sure that the WHERE clause is specified, and than matches it to the pattern discussed in the previous subsection. The, the satisfaction condition checks that the *AND* keyword separates the lower and upper value of the interval, checks that the constants are of the appropriate type and also checks the *?n* variable is bound to appropriate constants.

```
(p 22
"If the BETWEEN operator is used in a search condition,
there must be two constants of the same type as the attribute used in the condition."
(and (bind '?w (slot-value ss 'where) bindings)
     (not (null '?w))
     (match '(?*d1 (?is ?a attribute-p) ??n "BETWEEN" ?v1 ??p ?v2 ?*d2) '?w bindings))
(and (attribute-in-from ss '?a)
     (equalp '?p "AND")
     (equal (find-type '?a) (find-type '?v1))
     (equal (find-type '?a) (find-type '?v2))
     (member '?n '(nil "NOT")))
"WHERE")
```

4.3.4 Compiling constraints

As said earlier, CBM is computationally very simple because it reduces student modeling to pattern matching. The conditions of constraints are patterns matched against the student's solution. Pattern matching is a simple, but potentially time-consuming operation, especially in situations when the number of patterns is large. It is therefore often done in AI systems by using RETE networks (Forgy 1982). A detailed discussion of the RETE pattern matching algorithm is beyond the scope of this report and we briefly give the fundamental ideas only. RETE networks are designed for many pattern/many object situation, typical for expert systems, where there is a large number of facts (objects) describing the current problem state, and a large number of rules (patterns). A RETE network is the result of compiling patterns so that all conditions that appear in many patterns are applied only once. RETE networks consist of a number of nodes, which apply the conditions on objects they are given.

The situation in SQL-Tutor is much simpler, because there is only one object (i.e. the student's SELECT statement) the constraints should be matched against. Therefore, we developed a modification of the RETE network suitable for use in SQL-Tutor. There are two networks that are generated on the basis of constraints: the relevance network contains the relevance conditions, and the other network contains the satisfaction conditions of constraints. The student's

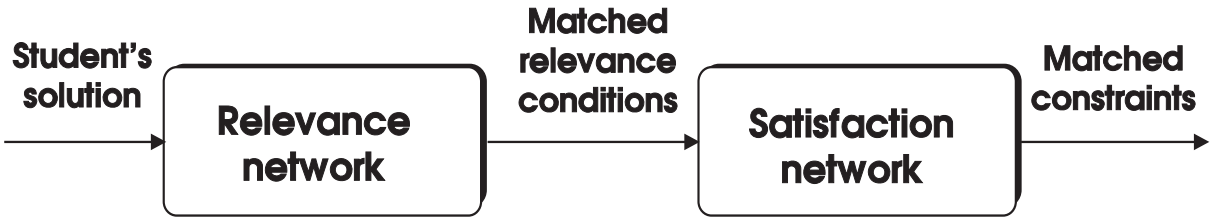

Figure 10: The compiled constraints

solution is first propagated through the relevance network (figure 10), and is propagated then through the satisfaction network, but only for those constraints that are relevant.

Relevance and satisfaction networks have slightly different structures[12]. There are three kinds of nodes in these networks: input, internal and output nodes. Each internal and input node is connected to a number of other nodes, referred to as its children[13]. Internal nodes of both networks are the same. Each internal node applies a test (corresponding to a condition in a constraint) to the student's solution (SS) and, if the test is satisfied, propagates SS to its children.

There are as many input nodes to the relevance network as there are different conditions which appear as first conditions in any constraint (currently 49). An input node in the relevance network applies a test to the student's solution, and if the test is satisfied, it propagates the students solution to its children. In the case where the result of the test is a list of bindings, it is also propagated to the children nodes. When SS reaches an output node in the relevance network, the corresponding constraint as added to the list of relevant constraints.

The input nodes to the satisfaction network contain the constraint number, and propagate SS to their children if the constraint has been found relevant in the first phase of matching. An output node of the satisfaction network adds the constraint number to the list of satisfied constraints. Appropriate binding lists are also stored, because they are used in feedback generation.

4.4 Student model

The structure of the student model is given below. In addition to the general information about the student (name and stereotype), the `student` class also maintains a history of the problems solved successfully by the student (the `problems` slot).

```
(defclass          student   ()
((name             :type     string   :accessor   student-name      :initarg   :name)
(solved-problems   :type     cons     :accessor   solved-problems   :initform  nil)
(knows             :type     cons     :accessor   knows             :initform  nil)
(stype             :type     atom     :accessor   stype             :initarg   stype)))
```

The `knows` slot is a list of structures that represent history information on the use of various constraints. For each constraint used by the students (in any problem the student worked on), there is a list of the following form:

(constr-no relevant used correct),

where *constr-no* is the unique constraint number, *relevant* is the indicator of how many times the constraint was found relevant for the ideal solutions, and *used* is the number of times the

[12]The class definitions are given in Appendix 1.

[13]The children of a node can either be internal or output nodes. The structures used in SQL-Tutor are trees, rather than networks, as every node can be a child of only one node.

constraint was relevant for the student's solution. *correct* is the number of times a relevant constraint was satisfied by the student's solution. These three indicators are used by PM for selecting new problems (as explained in 4.5) and are updated by the student modeler as follows.

After propagating a student's answer through the relevance and satisfaction networks, CBM comes up with two lists of constraints. The first list contains all the satisfied constraints, while the other list contains the violated ones. The constraints from the second list are then used by PM for generating feedback to the student, as discussed in 4.5. CBM uses these two lists to update the student model. If a constraint is satisfied, the *used* and *correct* indicators are incremented, and if the same constraint is relevant to the ideal solution as well, the *relevant* indicator is also incremented. For violated constraints, the only indicator that is incremented is *used* (potentially, if the constraint is relevant to the ideal solution, the *relevant* indicator is incremented).

4.5 Pedagogical module

The pedagogical module is the heart of the system; it selects problems to be given to the student and generates appropriate instructional actions according to the student model. The main goal of ITSs is the individualization of instruction. In SQL-Tutor, instruction can be individualized in several ways, by generating feedback dynamically, selecting topics and problems, and fading the scaffolding on the basis of the student model.

4.5.1 Feedback generation

The level of feedback determines how much information is provided to the student. Currently, there are five levels of feedback in SQL-Tutor, arranged below in the increasing order of amount of information: positive or negative feedback, error flag, hint, partial solution and complete solution.

At the lowest level (positive/negative feedback), the message simply informs the student whether the solution is correct (figure 11) or not and, in the later case, how many errors there are (figure 12). An error flag message informs the student about the clause[14] in which the error occurred (figure 13).

A hint-type message gives more information about the type of error, as illustrated in figure 14. Here, the student is given a general description of the cause of the error. This description is directly taken from the definition of constraint (the value of the *descr* slot of the corresponding constraint). Partial solution feedback displays the correct content of the clause in question, while the complete solution simply displays the correct solution of the current problem.

It was stated earlier that a student's solution may violate several constraints at the same time (as in figure 1, where 5 constraints were violated simultaneously). In such cases, SQL-Tutor examines the violated constraints and selects the one which is likely to be a genuine misconception. That is, SQL-Tutor select the constraint with the maximum difference between the *used* and *correct* indicators. The rationale here is the student has made the same error several times, and therefore instruction must start with that constraint. Currently, the student is told the total number of violated constraint, but the error messages only deal with one constraint at the time. This decision is based on our intuition that it would be much easier for the student to deal with errors one at the time. Furthermore, as constraints are modular, very often one misconception will cause the violation of several related constraints. It could possibly be quite puzzling for the student to see all the error messages at one; if, however, the student is able to correct the error on the basis of error messages from the first constraint, it may be the case that

[14] In case that there are several messages in various classes, the pedagogical module will select one of them to start with.

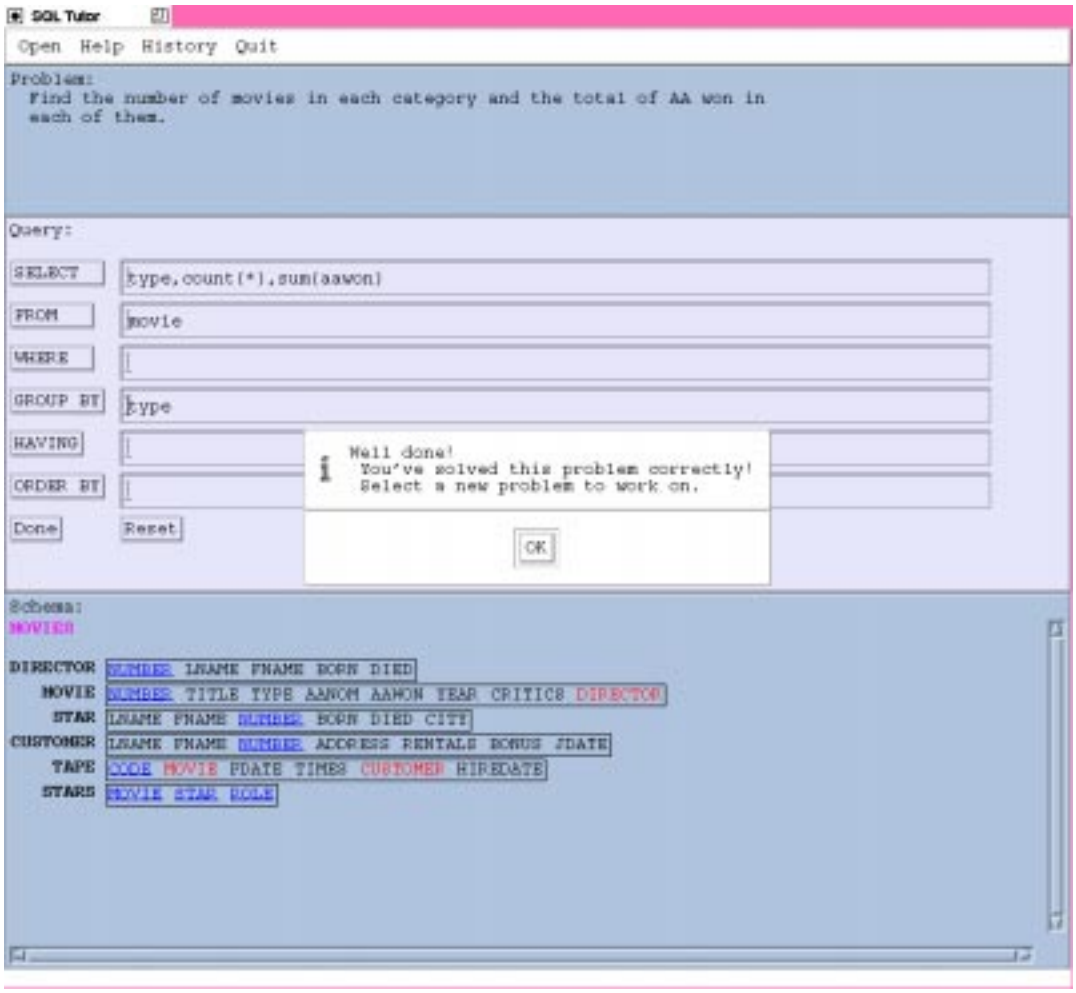

Figure 11: Correct solution

the other constraints are corrected at the same time, so the total number of violated constraints may be much smaller in the corrected solution.

4.5.2 Problem selection

Problems are also selected on the basis of a student model. SQL-Tutor examines the student model and selects a problem for a constraints that the student is not sure about (i.e., the one with the maximum `used - correct`). Another possibility for the problem to be posed to the student is a problem that requires the use of a constraint not present in the student model, as we want students to practice those constraints that they have not mastered. SQL-Tutor also allows the student to select the problem on his/her own. Such an approach introduces randomness in the coverage of constraints, which can mean that the student in practising the use of some known constraint or even introducing new ones. The randomness thus provides for challenge and/or review, and at the same time helps control for potential inaccuracies in the student model. Admittedly, the problem selection strategies just discussed are too simple and we are currently developing more sophisticated ones.

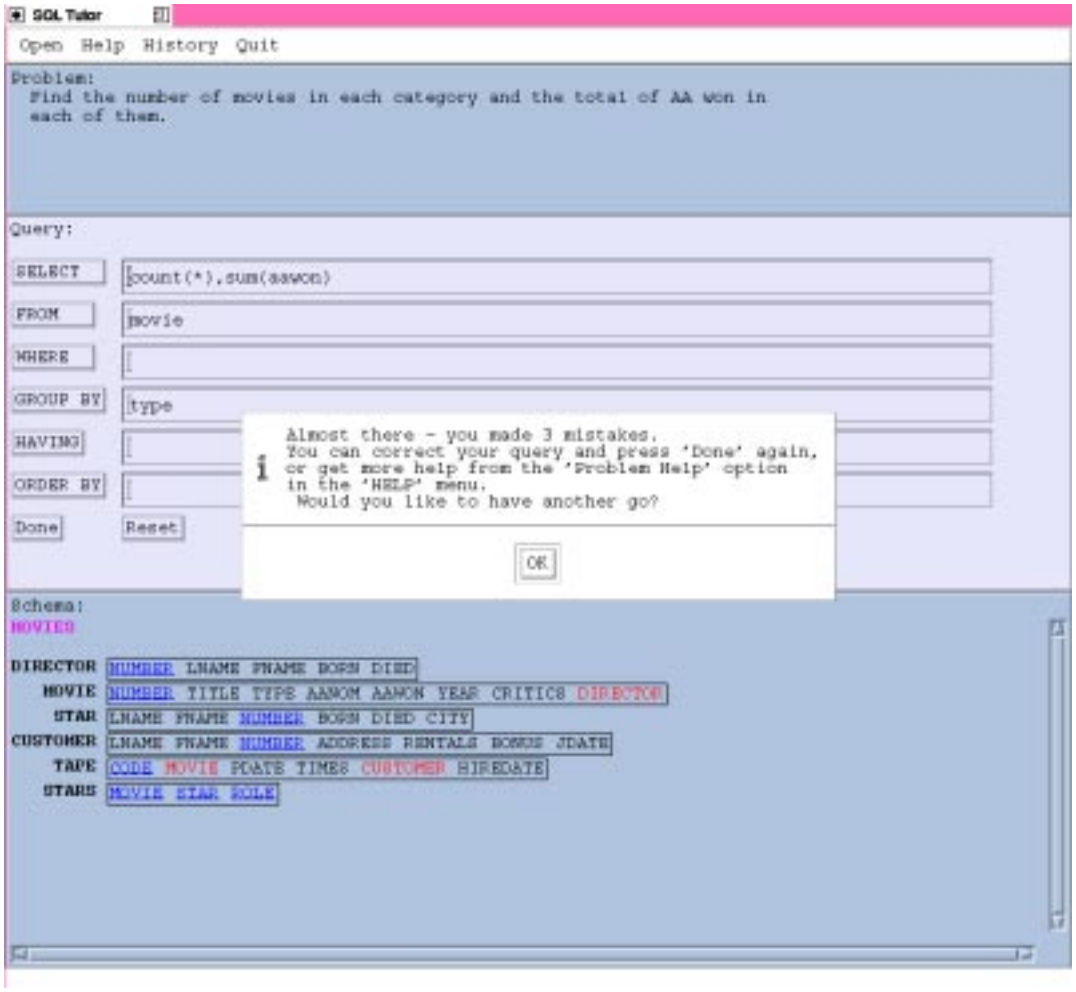

Figure 12: Negative feedback

4.5.3 Fading of scaffolding

Scaffolding is an educational concept that refers to providing support for students so that they can engage in activities that are normally beyond their abilities (Jackson et al. 1996). Scaffolding requires the instructional processes to be ground in prior experience and knowledge that students have. New representations and concepts being learnt must be related to students' existing knowledge, by using examples, analogies and multiple visual representations. Actions, effects and understanding must be coupled in a system in order for students to be able to learn; in other words, the system must provide immediate feedback to guide students and be able to answer to 'what-if' questions.

Fading of scaffolding subsumes the gradual removal of explicit support for a cognitive process (VanLehn 1996). At the beginning, a novice needs lots of support in order to deal with the cognitive load. However, it is necessary to fade the scaffolding in order to prevent the student from becoming too dependent on the feedback.

Scaffolding exists in two forms in SQL-Tutor, the user interface and the feedback, and both should be faded. Currently, SQL-Tutor does not support this, but it is straightforward to implement it. For example, it is possible to refuse the kinds of conceptual help requests if the student models documents that the student already possesses the requested knowledge. Also, as the student progresses through the problems, some kinds of feedback should become unavailable. For example, an experienced student should not request complete solutions of problems (s)he is

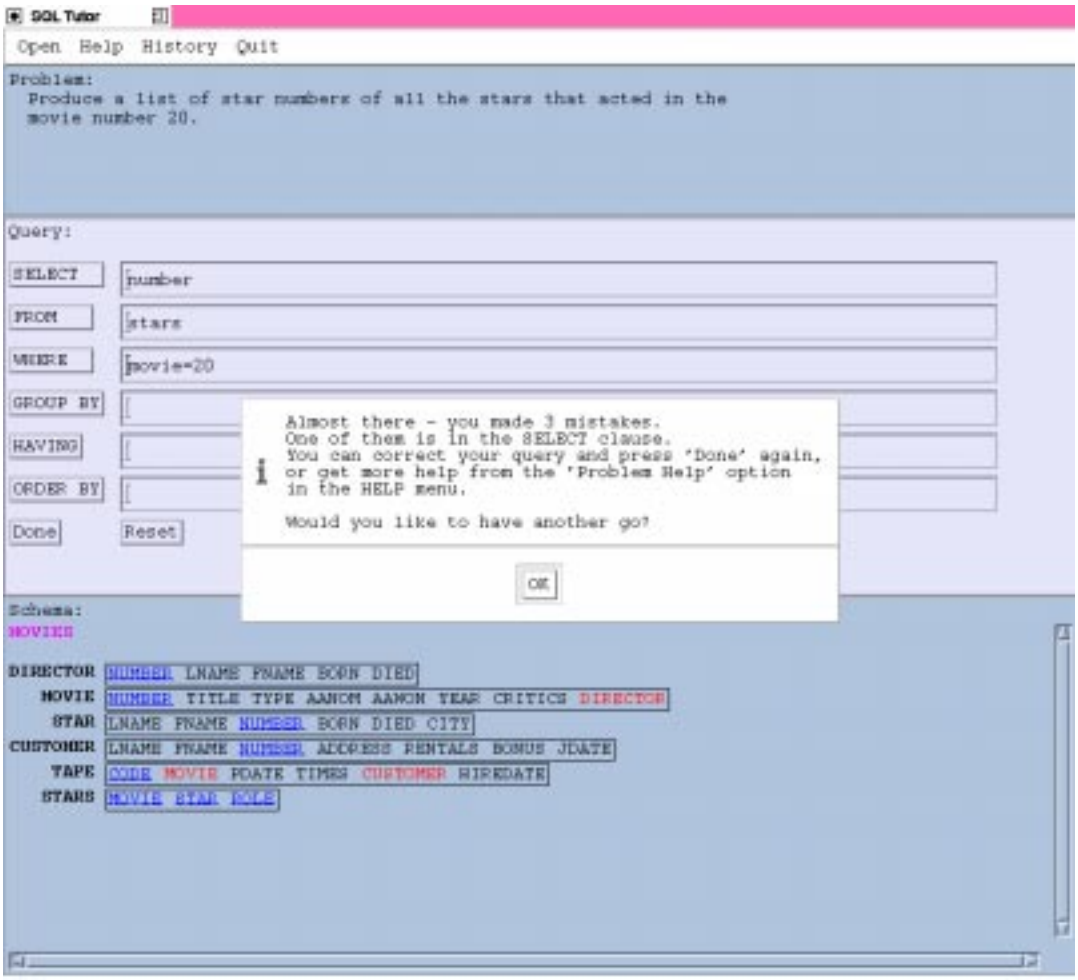

Figure 13: An error flag message

able to solve.

4.6 Learning in SQL-Tutor

SQL-Tutor is based on guided discovery and learning-by-doing. It supports three kinds of learning: conceptual, problem solving and meta-learning. The student can learn about concepts and elements of SQL by asking for explanations, using menu options and interface controls. SQL-Tutor is a problem-solving environment which supports acquisition of domain knowledge in a declarative form (i.e. constraints) and strengthening of this knowledge in practice. SQL-Tutor provides assistance in problem solving and arguments against incorrect actions. Finally, SQL-Tutor encourages meta-learning by supporting self-explanation on the basis of error messages and correct solutions. We plan to elaborate the pedagogical actions that will provide more emphasis on self-explanation, and also to incorporate other forms of meta-learning, such as using analogies.

5 Where next?

This report presented the current state in the implementation of SQL-Tutor. Before the system can be evaluated, there is a number of short-term goals. The interface should be completed; there are quite a few ideas which could greatly enhance the interface. For example, the student should

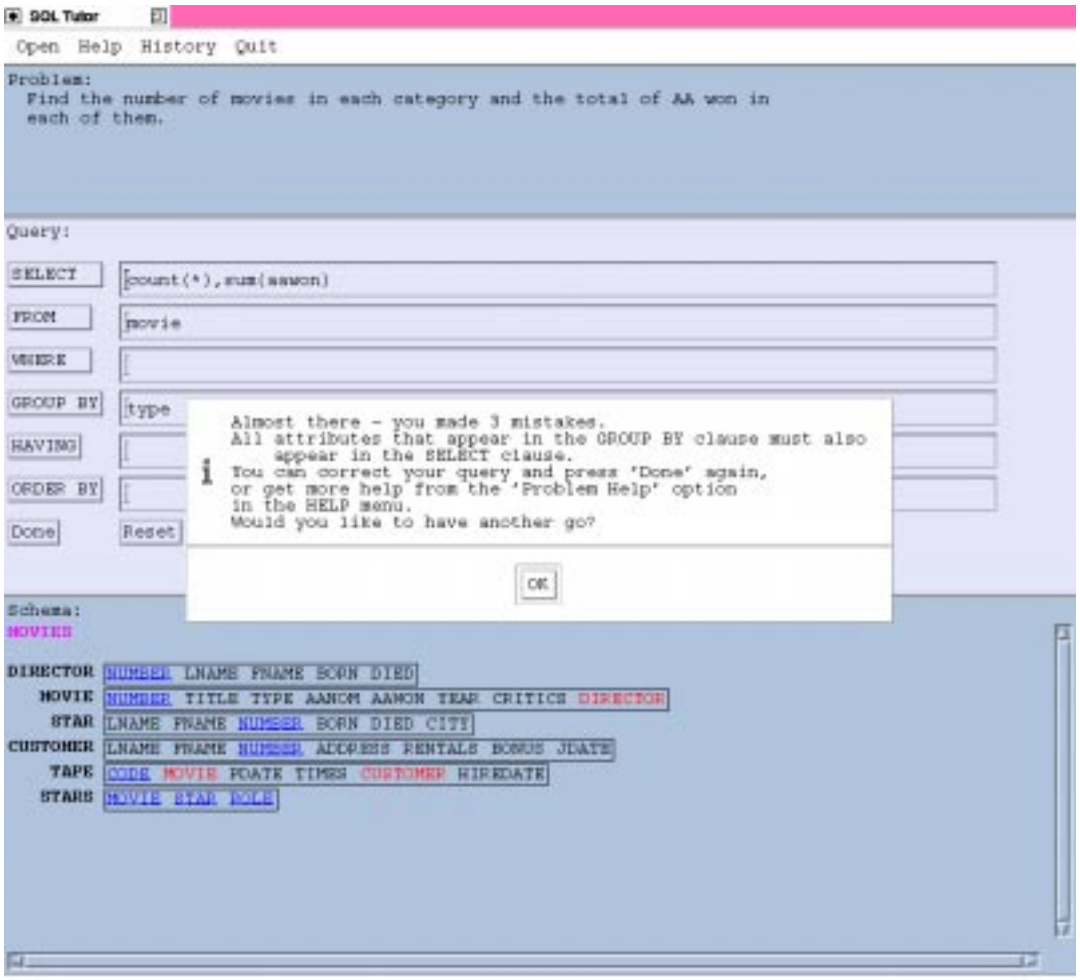

Figure 14: A hint displayed to the student

be able to select attribute/table names directly from the schema, instead of having to type them. It could be done with the introduction of a tool set, which could be used to select elements of the schema when the student wants to specify a join condition, or a selection condition, or simply enter names into the SELECT statement. At the same time, this would broaden the application of the equal opportunity to SQL-Tutor and hence make the system more student-friendly. The constraint base must also be widened and thoroughly tested. Currently, the state constraints is not complete and not tested widely enough.

Some of the directions for future research are more involved. We have already discussed the need to fade scaffolding and to support meta-learning. Currently, SQL-Tutor selects new problems in a primitive way, as explained in 4.5.2. Another way of selecting problems which would pay more attention to the concepts of the domain would be to introduce a curriculum. The curriculum would be composed of topics, and topics are linked to constraints. There are prerequisites for each topic. Such a structure would enable more meaningful selection of new problems.

In order to provide a more realistic working environment, we plan to connect SQL-Tutor to a DBMS. In such a way, the student may look at the contents of tables or query results. Some of the more involved goals are the design of instruction and fading of scaffolding, which were discussed in section 4.6.

6 Conclusions

The system has been shown to a number of database teachers, and all were very supported and expressed great enthusiasm for using it in their own courses. Directions for further improvements of the system were presented in 6. We believe that the system will be ready for classroom use in early 1998. It would be possible then to evaluate the interface and the effectiveness of CBM.

There are many possibilities for extending this research. There are many related areas in the database world, such as relational algebra and calculus, data modeling or normalization, which could serve as domains for other small instructional tools and be connected with SQL-Tutor into a database exploration "world".

One of the goals of this research is to put CBM to a real test. If CBM proves to be as good as it looks now, an authoring environment may be implemented on the basis of SQL-Tutor.

Acknowledgements

The work presented here was supported by the University of Canterbury research grant U6242 and by Computer Science Department grants. We thank Kendrick Elsey for implementing most of the interface.

References

1. Anderson J.R.: 1983, 'Acquisition of Proof Skills in Geometry'. In: R.S. Michalski, J.C. Carbonell and T.M.Mitchell (eds.): *Machine Learning: an Artificial Intelligence Approach*, Vol. 1. Tioga, Palo Alto, 191-219.
2. Anderson, J.R., Jeffries, R.: 1985, 'Novice LISP Errors: Undetected Losses of Information from Working Memory'. *Human–Computer Interaction* **22**, 403-423.
3. Anderson, J.R.: 1993, *Rules of the Mind*, Hillsdale, NJ: Lawrence Erlbaum Associates.
4. Anderson, J.R, Corbett, A.T., Koedinger, K.R., Pelletier, R.: 1995, 'Cognitive Tutors: Lessons Learned'. *The Journal of the Learning Sciences* **4**(2), 167-207.
5. Beck, J., Stern, M., Woolf, B.P.: 1997, 'Using the student model to control problem difficulty'. In: A. Jameson, C. Paris, and C. Tasso (eds), *Proc. of UM'97*, 277-288.
6. Bloom, B.: 1984, 'The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring'. *Educational Researcher* **13**, 3-16.
7. Brown, J.S., VanLehn, K.: 1980, 'Repair Theory: a Generative Theory of Bugs in Procedural Skills'. *Cognitive Science* **4**, 379-426.
8. Elmasri, R., Navathe, S.B.: 1994, *Fundamentals of database systems* Redwood: Benjamin/Cummings.
9. Forgy, C.L.: 1982, 'Rete: a fast algorithm for the many pattern/many object pattern match problem'. *Artificial Intelligence* **19**, 17-37.
10. Franz Inc.1996, *Allegro Common Lisp*.
11. Greer, J.E., McCalla, G.I. (eds.): *Student Modeling: the Key to Individualized Knowledge–based Instruction*. Berlin: Springer–Verlag, NATO ASI Series, 1994.
12. Holt, P., Dubs, S., Jones, M., Greer, J.E.: 1994, 'The State of Student Modelling'. In: J.E. Greer and G.I. McCalla (eds.): *Student Modeling: the Key to Individualized Knowledge–based Instruction*. Berlin: Springer–Verlag, NATO ASI Series, 3-35.
13. Jackson, S.L., Stratford, S.J., Krajcik, J., Soloway, E.: 1996, 'A Learner-Centered Tool for Students Building Models'. **CACM**, April 96.

14. Kearns, R., Shead, S., Fekete, A.: 1997, 'A teaching system for SQL'. In: Proc. of *Australasian Computer Science Education* ACSE'97, ACM Press, 224-231.
15. Norvig, P.: 1992, *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, San Mateo, CA: Morgan Kaufmann.
16. Ohlsson, S: 1994, 'Constraint–based Student Modeling'. In: J.E. Greer and G.I. McCalla (eds.): *Student Modeling: the Key to Individualized Knowledge–based Instruction.* Berlin: Springer–Verlag, NATO ASI Series, 167-189.
17. Ohlsson, S: 1996, 'Learning from Performance Errors'. *Psychological Review* **103**(2), 241–262.
18. Pratt, P.J.: 1990, *A Guide to SQL*, Boston: Boyd & Fraser.
19. Rossenbloom, P.S., Laird, J.E., Newell, A., R. McCarl: 1991, 'A preliminary analysis of the SOAR architecture as a basis for general intelligence'. *Artificial INtelligence* **47**, 289-305.
20. Self, J. A.: 1990, 'Bypassing the intractable problem of student modeling'. In: C. Frasson and G. Gauthier (eds.), *Intelligent Tutoring Systems: at the Crossroads of Artificial Intelligence and Education.* Norwood: Ablex, 107-123.
21. Self, J.: 1994, 'Formal Approaches to Student Modeling'. In: J.E. Greer and G.I. McCalla (eds.): *Student Modeling: the Key to Individualized Knowledge–based Instruction.* Berlin: Springer–Verlag, NATO ASI Series, 295-352.
22. Stern, M.; Beck, J., Woolf, B.P.: 1996, 'Adaptation of Problem Presentation and Feedback in an Intelligent Mathematics Tutor'. In: C. Frasson, G. Gauthier, A. Lesgold (eds.): *Intelligent Tutoring Systems.* New York: Springer-Verlag, 603-613.
23. Thimbleby, H.: 1990, *User Interface Design*, Reading, MA: ACM Press.
24. VanLehn, K.: 1988, 'Toward a Theory of Impasse–Driven Learning'. In: H. Mandl and A. Lesgold (eds.): *Learning Issues for Intelligent Tutoring Systems.* New York: Springer-Verlag, 19-41.
25. VanLehn, K.: 1996, 'Conceptual and Meta Learning During Coached Problem Solving'. In: C. Frasson, G. Gauthier, A. Lesgold (eds.): *Intelligent Tutoring Systems.* New York: Springer-Verlag, 29-47.
26. Wenger, E.: 1987, *Artificial Intelligence and Tutoring Systems.* Los Altos: Morgan Kaufmann.
27. Woolf, B.P., Murray, T.: 1994, 'Using Machine Learning to Advise a Student Model'. In: J.E. Greer and G.I. McCalla (eds.): *Student Modeling: the Key to Individualized Knowledge–based Instruction.* Berlin: Springer–Verlag, NATO ASI Series, 127-146.

Appendix 1: SQL-Tutor Classes

This appendix contains the definitions of all classes in SQL-Tutor. There are currently 13 classes defined, but this number will increase once the curriculum has been designed.

The `constr-kb` class is the class for the constraint base. It contains a description and a list of constraint numbers:

```
(defclass  constr-kb  ()
((descr    :type      string  :accessor  descr  :initarg   descr)
(lc        :type      cons    :accessor  lc     :initform  nil)))
```

The constraints themselves are instances of the `constraint` class, discussed in 4.3.

```
(defclass     constraint  ()
((number      :type       integer  :accessor  number       :initarg  number)
(rel-pattern  :type       cons     :accessor  rel-pattern  :initarg  rel-pattern)
(sat-pattern  :type       cons     :accessor  sat-pattern  :initarg  sat-pattern)
(problems     :type       cons     :accessor  problems     :initarg  problems)
(descr        :type       string   :accessor  descr        :initarg  descr)
(clause       :type       string   :accessor  clause       :initarg  clause)))
```

The relevance and satisfaction conditions of constraints are compiled into networks in order to speed up the matching process. The main structure is an instance of the `RETE-network` class, defined as:

```
(defclass             RETE-network  ()
((input-rel-nodes     :type         cons  :accessor  input-rel-nodes)
(matched-rel-con      :type         cons  :accessor  matched-rel-constrs)
(rel-bindings         :type         cons  :accessor  rel-bindings)
(input-sat-nodes      :type         cons  :accessor  input-sat-nodes)
(matched-sat-con      :type         cons  :accessor  matched-sat-constrs)
(sat-bindings         :type         cons  :accessor  sat-bindings)
(failed-sat-con       :type         cons  :accessor  failed-sat-constrs)
(failed-sat-bindings  :type         cons  :accessor  failed-sat-bindings)))
```

The *input-rel-nodes* slot contains all input nodes for the relevance network. The *matched-rel-con* slot contains a list of numbers of all matched relevance conditions, while the *rel-bindings* slot contains the corresponding binding lists. The *input-sat-nodes* slot contains all input nodes for the satisfaction network. The following two slots contain the list of all matched satisfaction conditions and corresponding binding lists. The last two slots contain the list of all violated constrains and corresponding binding lists.

All input nodes to the relevance network and the test nodes have the same structure, as defined by the `RETE-node` class. A node contains a test that is applied to the student solution (an element of a relevance or a satisfaction condition), and the list of all nodes that the solution should be propagated to if the test is satisfied (the *children* slot).

```
(defclass  RETE-node  ()
((test     :type      cons  :accessor  test      :initarg   test)
(children  :type      cons  :accessor  children  :initform  nil)))
```

Input nodes to the satisfaction network have a different structure, as they must be activated only if the relevance condition of the appropriate constraint is matched. The `input-sat-node` class inherits from the `RETE-node` class, and adds another slot to store the constraint number.

```
(defclass     input-sat-node  (RETE-node)
((constr-num  :type           integer      :accessor  constr-num  :initarg  constr-num)))
```

Finally, the output nodes need to store the number of the constraint in addition to the slots defined by the `RETE-node` class.

```
(defclass output-node (RETE-node)
((matched-constr :type cons :accessor matched-constr :initarg matched-constr)))
```

SQL-Tutor also contains the `problem` class, which stores the number of a problem, its text and target constraints (i.e., the constraints that are relevant to the ideal solution of the problem). Currently, there are 55 problems defined over three databases. New problems can be defined easily; there is a file that contains all the problems and their solution for each database and new problems can be added to the appropriate file.

```
(defclass       problem  ()
((no            :type    integer  :accessor  problem)
(text           :type    string   :accessor  text)
(target-constr  :type    cons     :accessor  target-constr)))
```

The `query` class is used to represent both student's and ideal solutions, which are distinguished by the value of the `type` slot. The class has 6 slots for each of the clauses of the SELECT statement, plus 6 slots that contain the actual strings the student entered. Finally, the `problem` slot contains the number of the problem to which the query applies.

```
(defclass   query  ()
((type      :type  atom    :accessor  type       :initarg  type)
(select     :type  cons    :accessor  select     :initarg  select)
(from       :type  cons    :accessor  from       :initarg  from)
(where      :type  cons    :accessor  where      :initarg  where)
(group-by   :type  cons    :accessor  group-by   :initarg  group-by)
(having     :type  cons    :accessor  having     :initarg  having)
(order-by   :type  cons    :accessor  order-by   :initarg  order-by)
(sel-txt    :type  string  :accessor  sel-txt    :initarg  sel-txt)
(from-txt   :type  string  :accessor  from-txt   :initarg  from-txt)
(where-txt  :type  string  :accessor  where-txt  :initarg  where-txt)
(group-txt  :type  string  :accessor  group-txt  :initarg  group-txt)
(hav-txt    :type  string  :accessor  hav-txt    :initarg  hav-txt)
(order-txt  :type  string  :accessor  order-txt  :initarg  order-txt)
(problem    :type  number  :accessor  problem    :initarg  problem)))
```

SQL-TUTOR also contains description of schemas (i.e. databases). The instances of this class are created from the SQL statements used to create the databases themselves in Ingres (given in Appendix 2). For each schema, there is a unique name, short description, a list of table names and a list of numbers of all problems defined for that database.

```
(defclass     schema   ()
((name        :type    string   :accessor   schema-name    :initarg    name)
(description  :type    string   :accessor   schema-descr   :initarg    :opis)
(tables       :type    cons     :accessor   tables        :initform   nil)
(problems     :type    cons     :accessor   problems      :initform   nil)))
```

SQL-TUTOR contains objects that represent tables (relations), as defined in the `relation` class. The objects of this class are created at the same time as schemas. A relation is represented with a unique name and the name of the database it belongs to. There is also a short description of the table, a list of attribute names and information about the primary key of the table.

```
(defclass     relation  ()
((name        :type     string   :accessor   rel-name    :initarg    :name)
(schema       :type     string   :accessor   db          :initarg    :schema)
(description  :type     string   :accessor   rel-descr   :initarg    :opis)
(attrs        :type     cons     :accessor   atributi    :initform   nil)
(primary-key  :type     cons     :accessor   pkey        :initform   nil)))
```

Attributes are represented as instances of the `attribute` class. For each attribute, there is a name (unique within the relation), a short description and type (the same as specified in the CREATE TABLE statement of SQL). The *required* slot specifies whether the attribute has been defined with the `NOT NULL` option or not. If the `UNIQUE` keyword was used to define the attribute, then the *unique* slot will evaluate to 't. Information about whether the attribute is a part of the primary or a foreign key is also being kept. Finally, the name of the relation the attribute belongs to is the last slot.

```
(defclass     attribute  ()
((aname       :type      string    :accessor   att-name     :initarg    :aname)
(descr        :type      string    :accessor   att-descr    :initarg    :opis)
(atype        :type      string    :accessor   type-of-att  :initarg    :atype)
(required     :type      boolean   :accessor   att-req      :initform   nil)
(unique       :type      boolean   :accessor   att-unique   :initform   nil)
(primary-key  :type      boolean   :accessor   att-pkey :    initarg     :apk)
(foreign-key  :type      string    :accessor   att-fkey     :initarg    :afk)
(relation     :type      string    :accessor   att-rel      :initarg    :arel)))
```

A student model is an instance of the `student` class. The slots of this class are discussed in 4.4.

```
(defclass          student   ()
((name             :type     string   :accessor   student-name     :initarg    :name)
(solved-problems   :type     cons     :accessor   solved-problems   :initform   nil)
(knows             :type     cons     :accessor   knows            :initform   nil)
(stype             :type     atom     :accessor   stype            :initarg    stype)))
```

The `stereotype` class is used to define student stereotypes. Currently, there are three stereotypes, representing beginners, intermediate and advanced students. They differ in the sets of initial constraint (the *init-constrs* slot contains the number of constraints which are deemed to be known by a student belonging to a stereotype).

```
(defclass      stereotype  ()
((name         :type       atom  :accessor  name          :initarg  name)
(init-constrs  :type       cons  :accessor  init-constrs  :initarg  init-costrs)))
```

Appendix 2: Databases

Currently there are three databases defined in SQL-TUTOR: *movies*, *registration* and *company*. This appendix gives the files containing the SQL statements used to create the databases, with additional comments about the tables which are stored as their descriptions[15].

1. **movies.def**

   "CREATE TABLE DIRECTOR
   (NUMBER /* Unique director's number */ INTEGER NOT NULL PRIMARY KEY,
   LNAME /* Last name */ VARCHAR(15) NOT NULL,
   FNAME /* First name */ VARCHAR(15) NOT NULL,
   BORN /* Year of birth */ INTEGER,
   DIED /* Year of death */ INTEGER);"
   "The DIRECTOR table holds information about all directors whose movies are held in the video club."

   "CREATE TABLE MOVIE
   (NUMBER /* Unique number for a movie */ INTEGER NOT NULL PRIMARY KEY,
   TITLE /* Title */ VARCHAR(30),
   TYPE /* Type of the movie */ VARCHAR(15) NOT NULL,
   AANOM /* Number of nominations for Academy Awards */ INTEGER,
   AAWON /* Number of AA won */ INTEGER,
   YEAR /* Year when the movie was made */ INTEGER,
   CRITICS /* Critics' rating */ VARCHAR(2),
   DIRECTOR /* Director's number */ INTEGER REFERENCES DIRECTOR);"
   "The MOVIE table contains information about all movies available in the club - their titles, directors, number of nominations for Academy Awards and the number of Awards won, critics' ratings and the director's number."

   "CREATE TABLE STAR
   (LNAME /* Last name */ VARCHAR(15) NOT NULL,
   FNAME /* First name */ VARCHAR(15) NOT NULL,
   NUMBER /* Unique number */ INTEGER NOT NULL,
   BORN /* Year of birth */ INTEGER,
   DIED /* Year of death */ INTEGER,
   CITY /* City of birth */ VARCHAR(10),
   PRIMARY KEY (NUMBER));"
   "The STAR table contains information about all actors/actresses that played in the movies available in the video club. This information includes: the star's name (first and last), number, city

[15] See the definition of the `relation` class in Appendix 1.

and year of birth and (if appropriate) year of death."

"CREATE TABLE CUSTOMER
(LNAME /* Last name */ VARCHAR(15) NOT NULL,
FNAME /* First name */ VARCHAR(15) NOT NULL,
NUMBER /* Unique number */ INTEGER NOT NULL,
ADDRESS /* Customer's address */ VARCHAR(40),
RENTALS /* The number of tapes rented */ INTEGER,
BONUS /* 1/10 of RENTALS */ INTEGER,
JDATE /* Date of joining the club */ DATE,
PRIMARY KEY (NUMBER));"
"The CUSTOMER table contains information about all customers: their numbers, names, addresses, number of rentals, bonuses and join dates."

"CREATE TABLE TAPE
(CODE /* Unique number */ INTEGER NOT NULL,
MOVIE /* Movie number */ INTEGER NOT NULL,
PDATE /* Purchase date */ DATE,
TIMES /* Times rented */ INTEGER,
CUSTOMER /* Number of the customer renting the tape */ INTEGER,
HIREDATE /* Date of hire */ DATE,
PRIMARY KEY (CODE),
FOREIGN KEY (CUSTOMER) REFERENCES CUSTOMER(NUMBER),
FOREIGN KEY (MOVIE) REFERENCES MOVIE,
CONSTRAINT CHECK_TIMES CHECK (TIMES ¿= 0));"
"The TAPE table contains information about all tapes owned by the video club. For each tape, there is information about its code, the movie it contains, purchase date, the number of times it was rented, the customer who has it at the moment and the hiredate."

"CREATE TABLE STARS
(MOVIE /* Movie number */ INTEGER NOT NULL REFERENCES MOVIE(NUMBER),
STAR /* Star number */ INTEGER NOT NULL REFERENCES STAR(NUMBER),
ROLE /* Name of the role */ VARCHAR(20) NOT NULL,
PRIMARY KEY (MOVIE,STAR,ROLE));"
"The STARS table specifies which star acted in what movie, with the name of the role played."

2. **company.def**

"CREATE TABLE DEPARTMENT
(DNAME /* Unique name of a department */ VARCHAR(15) NOT NULL UNIQUE,
DNUMBER /* Unique number of a department */ INTEGER NOT NULL PRIMARY KEY,
MGR /* IRD number of department's manager */ CHAR(9) NOT NULL,
MGRSTARTDATE /* Starting date of manager */ DATE);"
"The DEPARTMENT table contains information about all departments in the company. Each department is described by its name (unique), number (unique), identifier of the department's manager, and the starting date of the manager."

"CREATE TABLE EMPLOYEE
(IRD /* Unique IRD number of an employee */ CHAR(9) NOT NULL PRIMARY KEY,
LNAME /* Last name */ VARCHAR(15) NOT NULL,
MINIT /* Middle initial */ CHAR,
FNAME /* First name */ VARCHAR(15) NOT NULL,

BDATE /* Birthdate */ DATE,
ADDRESS /* Address */ VARCHAR(30),
SEX /* Employee's sex */ CHAR,
SALARY /* Salary */ DECIMAL(10,2),
SUPERVISOR /* IRD number of the supervisor */ CHAR(9),
DNO /* Number of the department the employee works for */ INTEGER REFERENCES DEPARTMENT);"
"The EMPLOYEE table contains information about all employees in a company. Each employee is described by his/her IRD number (unique), name (last name, middle initial and first name), address, sex, salary, identifier of the supervisor, and the number of the department the employee is working for."

"CREATE TABLE DEPT_LOCATIONS
(DNUMBER /* Unique department number */ INTEGER NOT NULL REFERENCES DEPARTMENT,
DLOCATION /* Department location */ VARCHAR(15) NOT NULL,
PRIMARY KEY (DNUMBER,DLOCATION));"
"The DEPT_LOCATIONS table contains information about all locations for each department."

"CREATE TABLE PROJECT
(PNAME /* Unique project name */ VARCHAR(15) NOT NULL UNIQUE,
PNUMBER /* Unique project number */ INTEGER NOT NULL PRIMARY KEY,
PLOCATION /* Project location */ VARCHAR(15),
DNUM /* Number of the controlling department */ INTEGER REFERENCES DEPARTMENT);"
"The PROJECT table holds information about all projects in the company. For every project, we know its name (unique), number (unique), location and the number of controlling department."

"CREATE TABLE WORKS_ON
(EIRD /* IRD number of the employee */ CHAR(9) NOT NULL,
PNO /* Project number */ INTEGER NOT NULL,
HOURS /* Number of hours the employee spends on the project */ DECIMAL(3,1) NOT NULL,
PRIMARY KEY (EIRD,PNO),
FOREIGN KEY (EIRD) REFERENCES EMPLOYEE,
FOREIGN KEY (PNO) REFERENCES PROJECT(PNUMBER));"
"The WORKS_ON table contains information about all projects employees are working on. It also contain the number of hours per project."

"CREATE TABLE DEPENDENT
(EIRD /* IRD number of the employee */ CHAR(9) NOT NULL REFERENCES EMPLOYEE,
DEPENDENT_NAME /* Dependent's name */ VARCHAR(15) NOT NULL, SEX /* Dependent's sex */ CHAR,
BDATE /* Dependent's birthdate */ DATE,
RELATIONSHIP /* Relationship with the employee */ VARCHAR(8),
PRIMARY KEY (EIRD,DEPENDENT_NAME));"
"The DEPENDENT table contains information about dependents of employees. A dependent is known by the identifier of the employee and his/her name. We also know dependent's sex, birthdate and the relationship with the employee."

3. **registration.def**

"create table VEHICLE_TYPE
(make /* Make of a vehicle */ varchar(10) not null,
model /* Model of a vehicle */ varchar(10) not null,
power /* Motive power (petrol, gas, diesel) */ char(1),
no_pass /* Number of passengers */ integer,
cap /* Capacity */ float,
cc /* Volume of the motor */ integer, primary key (make,model));"
"The VEHICLE_TYPE table stores information about various types of vehicles: their makes, models, motive power, the number of passengers, capacity and the volume of the motor."

"create table VEHICLE
(plates /* Plate number */ varchar(6) not null primary key,
year /* Year of manufacture */ char(4),
eng_no /* Engine number */ varchar(9),
ch_no /* Chassis number */ varchar(7) not null,
type /* Type of the vehicle (taxi, private, truck, ...) */ char(1),
make /* Make of a vehicle */ varchar(10),
model /* Model of a vehicle */ varchar(10),
foreign key (make) references vehicle_type,
foreign key (model) references vehicle_type);"
"The VEHICLE table stores information about vehicles: plate number, year of manufacture, engine and chassis number, type, make and model."

"create table EMPLOYEE
(fname /* Employee's first name */ varchar(15) not null,
init /* Employee's middle initial */ char(1),
lname /* Employee's last name */ varchar(15) not null,
IRD /* Employee's IRD number */ varchar(10) not null primary key,
sex /* Employee's sex */ char(1),
bdate /* Employee's birthdate */ char(10),
office /* Employee's office */ varchar(5),
reg_org /* The number of the registration office the employee works for */ varchar(10),
sdate /* Starting date in the organization */ char(10));"
"The EMPLOYEE tables stores information about all employees in registration organizations. For each employee, we know his/her name (first, middle initial and last), IRD number, sex, birthdate, office number, the number of the organization the employee is working for and the starting date."

"create table OWNER
(dr_lic /* Driver's licence number */ varchar(6) not null primary key,
IRD /* IRD number of the owner */ varchar(10),
fname /* Owner's first name */ varchar(15) not null,
init /* Middle initial */ char(1),
lname /* Owner's last name */ varchar(15) not null,
address /* Owner's address */ varchar(30) not null,
bdate /* Owner's birthdate */ char(16),
sex /* Owner's sex */ char(1),
emp /* Opis */ varchar(30) references employee,
phone /* Owner's phone number */ varchar(15));"
"The OWNER table stores information about owners of vehicles, such as the driver licence number, the IRD number, name (first, middle initial and last), address, phine number, burthdate, sex and

the number of the employee."

"create table OWNS
(plates /* Owner's plates number */ varchar(6) not null references vehicle,
ownerid /* Owner's IRD number */ varchar(6) not null references owner,
date /* The date of purchase */ char(10),
drr /* The mileage */ char(6),
primary key (plates,ownerid));"
"The OWNS table stores information about the owners of vehicels. It stores plate numebr, the owner's drivers licence number, the date when the vehicle was bought and the mileage at the time. "

"create table COLOR
(plates /* The plate number */ varchar(6) not null references vehicle,
color /* Color of the vehicle */ varchar(10) not null,
primary key (plates,color));"
"The COLOR table stores information about colors of vehicles. "

"create table REG_ORG
(number /* The number of the registration organization */ varchar(10) not null primary key,
street /* Street name */ varchar(15) not null,
st_num /* Number in the street */ varchar(6) not null,
city /* City */ varchar(10) not null,
manager /* The manager's IRD number */ varchar(10) references employee);"
"The REG_ORG table stores information about registration orgranizations. "

"create table FIRST_REG
(plates /* Plates */ varchar(6) not null references vehicle,
emp /* IRD of the employee who registered the vehicle */ varchar(10) not null references employee,
reg_org /* Organization number */ varchar(10) not null references reg_org,
reg_date /* Registration date */ char(10),
country /* The country */ varchar(10),
status /* Status of the vehicle */ char(1),
drr /* mileage */ char(6),
amount /* the price */ char(7),
primary key (plates,emp,reg_org));"
"The FIRST_REG table stores information about the first registration of a vehicle. "

"create table REGISTRATION
(PLATES /* Plates */ varchar(6) not null references vehicle,
emp /* IRD of the employee who registered the vehicle */ varchar(10) not null references employee,
reg_org /* Registration organization */ varchar(10) not null references reg_org,
reg_date /* Registration date */ char(10),
amount /* The price */ char(7),
primary key (plates,emp,reg_org));"
"The REGISTRATION table stores information about all registrations of vehicles. "